TABLE DES MATIERES

1. Introduction 1

Conception du manuel 3

Première exploration de GFA-BASIC 3.0 6

L'éditeur 9
- Notions de base 9
- Le bloc de touches du curseur 9
- Le bloc des touches numériques 11
- Autres touches d'éditions sur le bloc principal du clavier 12
- Instructions Control 12
- La barre des menus et les touches de fonction 14

Particularités 22
- DEFBIT, DEFBYT, DEFINT, DEFWRD, DEFFLT, DEFSTR 22
- DEFLIST 23
- $ 24

2. Variables et gestion de la mémoire 25

Types de variables 25
- !, |, &, %, #, $ 25

Tableaux 26
- DIM, DIM? 27
- OPTION BASE 28
- ARRAYFILL 29

Conversion de type 29
- TYPE 29
- ASC(), CHR$(), ... 30
- STR$() 31
- BIN$(), OCT$(), HEX$() 31
- VAL(), VAL?() 32
- CVx(), MKx$() 33
- CINT(), CFLOAT() 34

Opérations de pointeur 35
- * 35
- PEEK, POKE, DPEEK, LPEEK, POKE 36
- DPOKE, LPOKE, SPOKE, SDPOKE, SLPOKE 36

BYTE{}, CARD{}, INT{}, LONG{}, {}, FLOAT{} 37
SINGLE{}, DOUBLE{}, CHAR{} 37
VARPTR, V:, ARRPTR, * 39
ABSOLUTE 40

Suppression et échange **40**
CLEAR, CLR, ERASE 41
SWAP 42
SSORT, QSORT 43
INSERT, DELETE 45

Variables réservées **46**
FALSE, TRUE, PI 46
DATE$, TIME$, TIMER, SETTIME, DATE$=, TIME$= 47
TIMER 48

Particularités **48**
LET 48
VOID, ~ 49

Gestion de la mémoire **50**
FRE 50
BMOVE 50
BASEPAGE, HIMEM 51
RESERVE 52
INLINE 53
MALLOC, MFREE, MSHRINK 54

3. Opérateurs **57**

Opérateurs arithmétiques **57**
+ - * / ^ DIV \ MOD 57
+ - 57

Opérateurs logiques **58**
NOT 59
AND 60
OR 61
XOR 62
IMP 63
EQV 64

Opérateur de chaînes de caractères **65**
+ 65

Opérateurs de comparaison 65
= == >= <= <> 65

Opérateur d'affectation 68
= 68

Hiérarchie des opérateurs 69
( ) 69

4. Fonctions numériques 71

Fonctions mathématiques 71
ABS, SGN 71
ODD, EVEN 72
INT, TRUNC, FIX, FRAC 73
ROUND 74
MAX, MIN 74
SQR 75
EXP, LOG, LOG10 75
SIN, COS, TAN, ASIN, ACOS, ATN, DEG, RAD, SINQ, COSQ 76

Générateur de nombres aléatoires 77
RND, RANDOM, RAND, RANDOMIZE 77

Arithmétique entière 79

Instructions et fonctions 79
DEC, INC 79
ADD, SUB, MUL, DIV 80
PRED(), SUCC() 81
ADD(), SUB(), MUL(), DIV(), MOD() 81

Opérations de bits 83
BCLR, BSET, BCHG, BTST 83
SHL, SHR, ROL, ROR 84
AND(), OR(), XOR(), IMP(), EQV() 86
SWAP() 87
BYTE(), CARD(), WORD () 87

5. Gestion de chaînes de caractères 89

LEFT$, RIGHT$ 89
MID$ (en tant que fonction) 90
PRED, SUCC 90

LEN, TRIM$....91
INSTR....92
RINSTR....92
STRING$, SPACE$, SPC....93
UPPER$....93
LSET, RSET, MID$ (en tant qu'instruction)....94

6. Entrée au clavier et sortie sur l'écran....97

INKEY$....97
INPUT....98
LINE INPUT....99
FORM INPUT, FORM INPUT AS....100

PRINT, WRITE, PRINT AT(), LOCATE....101
PRINT USING, PRINT AT() USING....102
MODE....104
DEFNUM....104
CRSCOL, CRSLIN, POS, TAB....105
HTAB, VTAB....105

Les instructions KEYxxx....106
KEYPAD....106
KEYTEST, KEYGET, KEYLOOK....107
KEYPRESS....108
KEYDEF....109

Entrées et sorties générales....111

Lignes de données....111
DATA, READ, RESTORE....111

Gestion de fichiers....113

Répertoires....114
DFREE(), CHDRIVE, DIR$, CHDIR....114
DIR, FILES....115
FGETDTA, FSETDTA....116
FSFIRST, FSNEXT....117
MKDIR, RMDIR....118

Fichiers....118
EXIST....118
OPEN....119
LOF(), LOC(), EOF(), CLOSE, TOUCH....120

NAME AS, RENAME AS, KILL 121
BLOAD, BSAVE, BGET, BPUT 122

Accès séquentiel et séquentiel indexé **123**
INP#, OUT# 123
INPUT$(#) 124
INPUT#, LINE INPUT# 124
PRINT#, PRINT# USING, WRITE# 125
STORE, RECALL 126
SEEK, RELSEEK 127

Accès sélectif **128**
FIELD AS, AT 129
GET #, PUT #, RECORD 130

Communication avec la périphérie **131**
Entrée et sortie octet par octet 131
INP(), INP?(), OUT, OUT#, OUT?() 131

Interfaces sérielle et MIDI **132**
INPAUX$, INPMID$ 132

Souris et joystick **133**
MOUSEX, MOUSEY, MOUSEK, MOUSE 133
SETMOUSE 134
HIDEM, SHOWM 134
STICK, STICK(), STRIG() 135

Imprimante **137**
LPRINT, LPOS(), HARDCOPY 137

Génération de sons **138**
SOUND, WAVE 138

7. Commandes du programme **141**

Instructions de décision **142**
IF THEN ELSE ENDIF 142
ELSE IF 143

Branchements multiples **145**
ON GOSUB 145
SELECT, CASE, DEFAULT, ENDSELECT, CONT 145

Boucles **148**
FOR, STEP, NEXT 149
REPEAT, UNTIL 150
WHILE, WEND 151
DO, LOOP 152
DO WHILE, DO UNTIL, LOOP WHILE, LOOP UNTIL 152
EXIT IF 154

Procédures et fonctions **154**
GOSUB, @, PROCEDURE, RETURN 155
LOCAL 156

@func, FUNCTION, RETURN x, ENDFUNC 157
DEFFN, FN 158

Branchements liés à des événements **159**
ON BREAK, ON BREAK CONT, ON BREAK GOSUB 159
ON ERROR, ON ERROR GOSUB, RESUME, RESUME NEXT 160
ERROR, ERR, ERR$, FATAL 161

Programmation des interruptions **162**
EVERY, EVERY STOP, EVERY CONT 162
AFTER, AFTER STOP, AFTER CONT 162

Divers **164**
REM, ', ! 164
GOTO 165
PAUSE, DELAY 165
END, EDIT, STOP 166
NEW 167
LOAD 167
SAVE, PSAVE 167
LIST, LLIST 168
CHAIN 168
RUN 169
SYSTEM, QUIT 169

Traitement des erreurs **170**
TRON, TRON#, TROFF 170
TRON proc, TRACE$ 171
DUMP 172

8. Graphisme 175

Instructions de définition 176
SETCOLOR, COLOR 176
DEFMOUSE 176
DEFMARK 178
DEFFILL 179
BOUNDARY 180
DEFLINE 181
DEFTEXT 182
GRAPHMODE 184

Instructions graphiques générales 184
CLIP 185
PLOT, LINE, DRAW 186
DRAW, DRAW (), SETDRAW 187
BOX, PBOX, RBOX, PRBOX 189
CIRCLE, PCIRCLE, ELLIPSE, PELLIPSE 190
POLYLINE, POLYMARK, POLYFILL 191
POINT() 191
FILL 192
CLS 193
TEXT 193
SPRITE 194

Sections d'écran 195
SGET, SPUT 195
GET, PUT 195
VSYNC 196
BITBLT 197

9. Gestions des événements, menus et fenêtres 201

Gestion des événements 201
ON MENU 201
MENU() 202
ON MENU BUTTON GOSUB 205
ON MENU KEY GOSUB 206
ON MENU IBOX GOSUB, ON MENU OBOX GOSUB 207
ON MENU MESSAGE GOSUB 208

Menus déroulants 209
ON MENU GOSUB, MENU m$() 209
MENU OFF, MENU KILL 210
MENU 211

Instructions de fenêtres 212
OPENW, CLOSEW 213
W_HAND, W_INDEX 214
CLEARW, TITLEW, INFOW, TOPW, FULLW 215
WINDTAB 216

Divers 218
RC_INTERSECT 218
RC_COPY TO 219
ALERT 219
FILESELECT 220

10. Routines système 223

GEMDOS, BIOS, XBIOS 223
L:, W: 224

Appels Line-A 225
ACLIP 225
PSET 225
PTST() 226
ALINE 226
HLINE 227
ARECT 228
APOLY TO 228
BITBLT 229
ACHAR 233
ATEXT 233
L~A 234

Appels VDI 234
CONTRL(), INTIN(), PTSIN(), INTOUT(), PTSOUT() 236
VDISYS 236
VDIBASE 237
WORK_OUT() 238

Routines VDI spéciales et GDOS 238
GDOS? 239
V~H 240
V_OPNWK, V_CLSWK 240
V_OPNVWK, V_CLSVWK 241
V_CLRWK, V_UPDWK 241
VST_LOAD_FONTS, VST_UNLOAD_FONTS 242
VQT_EXTENT 243
VQT_NAME 243

Appel de routines d'autres langages **244**
C: 244
MONITOR 246
CALL 246
RCALL 247
EXEC 248

11. Bibliothèques AES **251**

GCONTRL, ADDRIN, ADDROUT, GINTIN, GINTOUT, GB 251
GEMSYS 252

Structure d'objet **253**
OB_NEXT, OB_HEAD, OB_TAIL 255
OB_TYPE, OB_SPEC 255
OB_STATE, OB_FLAGS, OB_X, OB_Y, OB_W, OB_H 255
OB_ADR 255
Structure d'informations de texte (TEDINFO) 256
Structure d'icône (ICONBLK) 256
Structure d'image de bits (BITBLK) 257
Structure de bloc d'application (USERBLK) 257
Structure de bloc de paramètres (PARMBLK) 257

Bibliothèque d'application (application library) **258**
APPL_INIT 258
APPL_READ 258
APPL_WRITE 259
APPL_FIND 259
APPL_TPLAY 260
APPL_TRECORD 260
APPL_EXIT 260

Bibliothèque d'événements (event library) **260**
EVNT_KEYBD 260
EVNT_BUTTON 261
EVNT_MOUSE 262
EVNT_MESAG 263
EVNT_TIMER 263
EVNT_MULTI 264
EVNT_DCLICK 265

Bibliothèques de menu (menu library) **265**
MENU_BAR 265
MENU_ICHECK 265
MENU_IENABLE 266
MENU_TNORMAL 266

MENU_TEXT 266
MENU_REGISTER 267

Bibliothèque d'objet (object library) 267
OBJC_ADD 267
OBJC_DELETE 268
OBJC_DRAW 268
OBJC_FIND 268
OBJC_OFFSET 269
OBJC_ORDER 269
OBJC_EDIT 270
OBJC_CHANGE 270

Bibliothèque de formulaire (form library) 271
FORM_DO 271
FORM_DIAL 271
FORM_ALERT 272
FORM_ERROR 272
FORM_CENTER 273
FORM_KEYBD 273
FORM_BUTTON 273

Bibliothèque graphique (graphics library) 274
GRAF_RUBBERBOX 274
GRAF_DRAGBOX 275
GRAF_MOVEBOX 275
GRAF_GROWBOX 276
GRAF_SHRINKBOX 276
GRAF_WATCHBOX 277
GRAF_SLIDEBOX 277
GRAF_HANDLE 278
GRAF_MOUSE 278
GRAF_MKSTATE 279

Bibliothèque Scrap (srap library) 279
SCRP_READ 279
SCRP_WRITE 280

Bibliothèque de sélecteur de fichier (file selector library) 280
FSEL_INPUT 280

Bibliothèque de fenêtre (window library) 281
WIND_CREATE 281
WIND_OPEN 282
WIND_CLOSE 282
WIND_DELETE 282
WIND_GET 283

WIND_SET 284
WIND_FIND 285
WIND_UPDATE 286
WIND_CALC 286

Bibliothèque Resource (resource library) **287**
RSRC_LOAD 287
RSRC_FREE 287
RSRC_GADDR 288
RSRC_SADDR 288
RSRC_OBFIX 289

Bibliothèque Shell (shell library) **289**
SHEL_READ 290
SHEL_WRITE 290
SHEL_GET 291
SHEL_PUT 291
SHEL_FIND 292
SHEL_ENVRN 292

Programmes d'exemple **293**

12. Annexes **301**

Compatibilité avec GFA-BASIC 2.xx **301**
Tables GEMDOS 303
Tables BIOS 312
Table XBIOS 314
Table des variables Line-A 322
Table des paramètres d'entrée pour V_OPN(v)WK 324
Table VT 52 326
Table des codes clavier et caractères ASCII spéciaux 327
Table des codes ASCII 328
Table de motifs de remplissage et styles de ligne 329

Messages d'erreur **331**
Messages d'erreur GFA-BASIC 331
Messages d'erreur bombes 332
Messages d'erreur du TOS 333
Messages d'erreur de l'éditeur 333

Liste alphabétique des fonctions **335**

Index **349**

SUPPORT PRODUIT

Seules les personnes retournant la carte client dûment remplie, en incluant bien nom, adresse, nom du produit et numéro de série, seront enregistrées comme client Micro Application et pourront bénéficier du support produit.

Nous rappelons que le support produit est effectué par l'équipe technique de Micro Application.

Les horaires sont :

Du Lundi au Jeudi : **14 h 30 à 17 h 30**

Le Vendredi : **14 h 30 à 16 h 30**

Toute personne n'ayant pas retourné chaque carte spécifique à chaque produit se verra refuser tout support.

1. INTRODUCTION

Vous disposez avec l'interpréteur GFA-BASIC 3.0 d'un langage de programmation très riche ainsi que d'un environnement de développement très pratique. Cette version très avancée du BASIC dispose d'un éditeur puissant et rapide qui permet la réalisation de programmes structurés.

L'interpréteur vous offre des possibilités extrêmement commodes de traitement des erreurs, à travers des instructions spéciales de recherche et d'élimination des erreurs (par exemple Tron procedure et TRACE$). Dans les langages de programmation modernes, les possibilités de structuration des programmes revêtent une importance particulière. L'éditeur soutient déjà la programmation structurée en opérant un retrait automatique des instructions figurant dans des boucles ou conditions. D'autre part, les sous-programmes peuvent être remplacés par des noms symboliques dans le listing du programme, le sous-programme pouvant ensuite être "déroulé" à l'emplacement du symbole en frappant simplement une touche.

Dans le domaine des instructions conditionnelles, les instructions de branchements multiples (ELSE IF, SELECT-CASE) sont venues s'ajouter aux instructions IF-ELSE-ENDIF qui existaient déjà dans les versions plus anciennes de GFA-BASIC.

Pour formuler des sous-programmes, il est possible de définir des procédures et fonctions (une innovation de 3.0) avec transmission de valeurs de variables ainsi que de variables elles-mêmes. Vous disposez aussi, outre les types de boucle déjà connus dans les anciennes versions de GFA-BASIC, FOR-NEXT, REPEAT-UNTIL, WHILE-WEND et DO-LOOP, d'instructions de boucle supplémentaires telles que DO-UNTIL, DO-WHILE, LOOP-UNTIL et LOOP-WHILE.

Une programmation proche du système est également autorisée grâce à la possibilité d'appeler les routines du système d'exploitation (GEMDOS, BIOS, XBIOS). Bon nombre de ces fonctions sont aussi disponibles sous forme d'instructions simples. La programmation pilotée par interruption est également possible avec EVERY et AFTER. Des sous-programmes en assembleur et en C peuvent être intégrés à l'aide d'instructions telles que RCALL, C: et Monitor, par exemple.

Les principales routines VDI et toutes les fonctions AES (par exemple les fonctions de gestion de menus, de fenêtres et de formulaires) sont disponibles sous forme de fonctions intégrées. Avec quelques instructions supplémentaires particulièrement simples pour l'intégration sous GEM, cela vous permet un développement très aisé des programmes qui doivent avoir une interface utilisateur GEM.

L'interpréteur GFA-BASIC 3.0 dispose d'une véritable arithmétique entière, qui permet une vitesse de calcul très élevée, ainsi que d'une arithmétique à virgule flottante de haute précision (13 chiffres après la virgule).

Par rapport aux versions 2.xx du GFA-BASIC, d'autres types de variables (BYTE, WORD) et opérations de bits (BCLR, BSET, BTST, BCHG, SHL, SHR, ROL, ROR, etc...) ont été ajoutés. Les programmes graphiques peuvent utiliser très facilement des routines LINE-A qui sont intégrées sous forme d'instructions.

CONCEPTION DE CE MANUEL

Ce manuel commence par une brève description du langage GFA-BASIC 3.0. Vient ensuite une explication de la structure du manuel (c'est le chapitre que vous avez justement sous les yeux) ainsi qu'une introduction à l'utilisation de l'interpréteur GFA-BASIC. Ce chapitre se termine par une description des points dont il faut tenir compte pour reprendre des programmes qui ont été écrits sous d'anciennes versions de GFA-BASIC. Le chapitre suivant décrit les instructions d'utilisation de l'éditeur. Tous les chapitres suivants contiennent essentiellement une description des instructions et fonctions de GFA-BASIC 3.0. Ces instructions et fonctions sont classées par catégories logiques, les notions apparentées étant expliquées en même temps (par exemple MIN et MAX). Vous trouverez aussi en annexe une liste des instructions classées par ordre alphabétique qui renvoie aux pages du manuel où sont traitées les instructions. L'explication de chaque instruction comprend les parties suivantes :

- *Indication de la syntaxe*
- *Description des types de paramètres autorisés*
- *Texte explicatif*
- *Exemple*

Dans l'indication de la syntaxe, les paramètres optionnels sont marqués par des crochets rectangulaires, par exemple :

```
LEFT$(a$ [,x] )
```

En GFA-BASIC, on distingue des instructions et des fonctions. Les instructions ne renvoient pas de valeur :

```
LINE 100,100,200,200
```

Les fonctions renvoient une valeur de réponse qui peut être sortie avec PRINT, affectée à une variable ou constituer un membre d'une expression. Voici quelques exemples :

```
PRINT ASC("65")
PRINT ASC("A")
a = ASC("A")
b = ASC("A") + 32
```

Dans ce manuel, lors de l'indication de la syntaxe des fonctions, nous ne précisons pas qu'elles renvoient une valeur de réponse car cela ressort de la description des instructions. Nous indiquerons par exemple simplement ASC(a$) pour décrire la syntaxe.

Les instructions qui permettent de spécifier un nombre illimité de paramètres (comme DATA par exemple) sont aussi suivies de crochets. Dans ce cas, les crochets contiennent deux paramètres, suivis de trois points, par exemple :

```
DATA [x,y,...]
```

L'indication des types de paramètres autorisés figure sous l'indication de syntaxe. Pour désigner ces types, nous utilisons les abréviations suivantes :

avar : *variable arithmétique* (arithmetic variable). Il doit s'agir d'une variable numérique d'un type quelconque.

aexp : *expression arithmétique* (arithmetic expression). Il s'agit d'une expression d'un degré de complexité quelconque, produisant un nombre. Il peut s'agir aussi bien d'une constante (c'est-à-dire d'un nombre) que d'une variable (les variables formant un sous-ensemble des expressions). Voici des exemples d'expressions arithmétiques :

```
a%
3
2+a%+ASC("A")
```

svar : *variable alphanumérique* (string variable). Il s'agit d'une variable de chaîne de caractères. Ce type de variable porte la marque *$*.

sexp : *expression alphanumérique* (string expression). Cette expression peut être d'un degré de complexité quelconque et elle doit produire une chaîne de caractères comme résultat. Il peut s'agir également d'une constante (un texte entre guillemets) aussi bien que d'une variable alphanumérique. Voici des exemples d'expressions alphanumériques :

```
a$
"Test"
a$+"Essai"+LEFT$("MANUEL",3)
```

ivar *variable entière* (integer variable).

iexp *expression entière* (integer expression).

bexp *expression logique* (boolean expression).

Il est important de noter que pour certains paramètres numériques il n'est pas possible de fournir n'importe quels types de variables numériques. Les adresses en sont l'exemple par excellence. Les adresses doivent être indiquées dans une variable d'au moins quatre octets ; les variables booléennes, d'octet ou de mot sont donc déplacées. La description des types de paramètres autorisés est suivie de l'explication des instructions, où est décrit le rôle de l'instruction et de ses différents paramètres.

L'explication d'une instruction se termine par un ou plusieurs exemples. Chacun de ces exemples peut être entré sous l'éditeur et lancé avec RUN (Shift+F10 ou en cliquant RUN dans la ligne de menu). L'effet produit est signalé à la suite de chaque exemple.

Ce principe de description des instructions n'a été quelque peu adapté que dans la section consacrée aux bibliothèques AES. Dans cette section en effet, nous indiquons le nom de l'instruction, suivi d'une description de sa fonction et ensuite seulement de la syntaxe de l'instruction avec l'explication des différents paramètres. Plusieurs longs programmes d'exemple vous sont fournis à la fin du chapitre consacré aux bibliothèques AES.

La raison de cette différence de présentation est que bon nombre des instructions de ce chapitre ne fonctionnent de façon intéressante qu'en liaison avec plusieurs autres instructions, de sorte que les programmes d'exemple doivent nécessairement contenir de nombreuses instructions AES et être suffisamment complets.

Ce manuel d'utilisation se conclut par un recueil de tables et par une liste alphabétique de toutes les instructions avec indication des numéros des pages où ces instructions sont décrites.

PREMIERE EXPLORATION DE GFA-BASIC 3.0

Cette section s'adresse aux utilisateurs de GFA-BASIC 3.0 qui n'ont encore aucune expérience dans ce langage de programmation. Ceux qui ont déjà travaillé avec les anciennes versions peuvent donc faire l'impasse sur cette section.

La disquette de programme de GFA-BASIC n'est pas protégée contre la copie. Commencez donc par réaliser une copie de sécurité de la disquette originale. Reportez-vous au manuel d'utilisation de votre ordinateur pour une description de l'opération de copie. Introduisez ensuite la copie de la disquette programme dans votre lecteur de disquette et lancez l'interpréteur GFA-BASIC (*"GFABASIC.PRG"*).

Sur l'écran apparaît maintenant l'éditeur, sous lequel vous pouvez écrire vos programmes. Entrez maintenant les lignes de programme ci-dessous et appuyez sur la touche *Return* après avoir entré chaque ligne.

Il n'est pas nécessaire d'entrer également les espaces qui figurent devant certaines instructions. Les retraits dans les instructions de boucles sont en effet gérés automatiquement par l'éditeur. Il n'est pas nécessaire non plus, lorsque vous entrez les lignes d'instructions, de distinguer les majuscules et les minuscules. Chaque fois que vous sortez d'une ligne avec *Return*, l'écriture en majuscules ou minuscules est automatiquement opérée.

```
DEFFILL 1,0
REPEAT
  WHILE MOUSEK=1
    PBOX MOUSEX,MOUSEY,MOUSEX+30,MOUSEY+30
  WEND
UNTIL MOUSEK=2
```

Dans le coin supérieur droit de l'écran se trouve le mot *"Run"*. Désignez ce mot avec la flèche de la souris et appuyez sur le bouton gauche de la souris pour lancer le programme.

Vous voyez maintenant apparaître un écran blanc, sur lequel se détache la flèche de la souris. Si vous appuyez maintenant sur le bouton gauche de la souris et déplacez la souris sur votre table, vous pouvez dessiner sur l'écran. C'est un rectangle qui est utilisé en guise de *"pinceau"*. Vous mettez fin au programme en appuyant sur le bouton droit de la souris. Vous voyez alors apparaître sur l'écran une boîte d'alerte contenant le message *"Fin du programme"*.

Désignez le mot *"Return"* de cette boîte et appuyez sur le bouton gauche de la souris. Vous vous retrouvez alors sous l'éditeur.

Comment fonctionne ce programme ? Le dessin est réalisé à l'aide de l'instruction *PBOX* au milieu du programme. Cette instruction dessine des rectangles pleins. Ses quatre premiers paramètres définissent les coins de ce rectangle. La première instruction de ce programme (DEFFILL 1,0) définit que le rectangle devra être rempli de blanc.

Les variables MOUSEK, MOUSEX et MOUSEY contiennent des informations sur la souris. MOUSEK indique quel bouton de la souris est actuellement actionné. MOUSEK=1 signifie qu'il s'agit du bouton de gauche, MOUSEK=2 qu'il s'agit du bouton de droite. MOUSEX et MOUSEY contiennent respectivement les positions x et y du point d'action de la souris sur l'écran (cf. Graphisme, DEFMOUSE).

Les autres instructions (REPEAT, WHILE, WEND, UNTIL) sont des instructions de boucle. La boucle composée des instructions WHILE MOUSEK=1 et WEND peut être traduite par la formule *"répéter tant que le bouton gauche de la souris est enfoncé"*. La boucle extérieure, composée des instructions REPEAT et UNTIL MOUSEK=2 signifie *"répéter jusqu'à ce que le bouton droit de la souris soit actionné"*. Comme la ligne avec l'instruction UNTIL n'est pas suivie d'autres instructions, le programme se termine dès que la boucle REPEAT-UNTIL est abandonnée.

Placez-vous maintenant après la dernière ligne du programme et entrez la ligne suivante, qui est volontairement incorrecte puisqu'il y manque le i du mot d'instruction PRINT :

```
prnt "Test"
```

Si vous appuyez maintenant sur la touche *Return* pour confirmer l'instruction et abandonner la ligne, un signal sonore retentit et le message *"Syntax Error"* apparaît sur la seconde ligne de l'écran.

L'éditeur contrôle donc dès la phase d'écriture du programme si la syntaxe des instructions entrées est correcte. Amenez maintenant le curseur sur la lettre R et appuyez trois fois sur la touche *Delete*. Vous ne voyez plus maintenant que :

```
p "Test"
```

sur la ligne de programme. Si vous appuyez maintenant sur la touche *Return*, vous pouvez abandonner la ligne et la lettre p sera automatiquement identifiée comme l'abréviation du mot d'instruction PRINT.

Nous n'irons pas plus loin pour cette première prise de contact avec l'interpréteur.

L'EDITEUR

QUELQUES NOTIONS DE BASE

L'éditeur du GFA-BASIC 3.0 n'est pas un éditeur de texte habituel car il a été conçu spécialement pour le développement de programmes. C'est ce qui explique par exemple que les instructions dont la syntaxe est incorrecte soient détectées dès l'écriture du programme. De même, les instructions figurant dans des boucles ou des instructions conditionnelles sont automatiquement écrites en retrait et les abréviations des instructions sont remplacées par les noms d'instructions in extenso (par exemple p par PRINT).

Lorsque vous écrivez un listing de programme, un contrôle de syntaxe est effectué chaque fois qu'il s'agit de valider une ligne. Si l'instruction figurant sur cette ligne est d'une syntaxe incorrecte, le message *"Syntax Error"* apparaît sur la seconde ligne. La ligne ne peut être abandonnée jusqu'à ce que l'erreur soit corrigée. Il est toutefois possible de quitter la ligne avant de la corriger, à condition de placer un signe de commentaire (') au début de la ligne.

Chaque ligne de programme ne peut recevoir qu'une seule instruction. Cette instruction peut encore être suivie d'un texte de commentaire. L'instruction ne doit pas comporter plus de 255 caractères. Lorsqu'une ligne dépasse les 80 caractères, le début de cette ligne disparaît du côté gauche de l'écran. La ligne subit donc un défilement ou *"scrolling"* horizontal.

Chaque fois qu'une ligne d'instruction est validée, un contrôle de sa syntaxe (voir plus haut) est effectué, l'instruction est éventuellement écrite avec le retrait qui convient et la ligne est formatée. Le formatage est une opération consistant à éliminer les espaces superflus (2 + 2 devient par exemple 2+2) et les mots d'instructions et noms de variables sont réécrits conformément au DEFLIST fixé. Le DEFLIST 0, qui est prédéfini, prévoit par exemple que tous les mots d'instructions soient écrits en majuscules et tous les noms de variables en minuscules.

LE BLOC DES TOUCHES DU CURSEUR

La gestion du curseur s'effectue à l'aide du bloc de touches curseur. Voici la signification des touches de ce bloc :

Flèche gauche	-->	Curseur un caractère vers la gauche
Flèche droite	-->	Curseur un caractère vers la droite
Flèche haut	-->	Curseur une ligne vers le haut
Flèche bas	-->	Curseur une ligne vers le bas

Les déplacements du curseur sont cependant soumis à certaines limites. Le curseur ne peut pas être amené plus loin qu'un caractère après le dernier caractère d'une ligne ni plus loin qu'une ligne en dessous de la dernière ligne de programme. Lorsque le curseur parvient sur une ligne plus courte que la position de colonne actuelle du curseur, le curseur saute à la fin de cette ligne, sinon il conserve sa position de colonne actuelle (contrairement à ce qui se passait sous les éditeurs des anciennes versions de GFA-BASIC).

Il est également possible de positionner le curseur avec la souris. Il faut pour cela désigner la position voulue avec la flèche de la souris puis appuyer sur le bouton gauche de la souris.

Lorsque la touche *Insert* est actionnée, une ligne vide est insérée entre la ligne au-dessus du curseur et la ligne du curseur à condition qu'aucune modification n'ait encore été effectuée sur la ligne actuelle. Le curseur est alors placé au début de cette ligne vide. *Clr/Home* amène le curseur dans le coin supérieur gauche en dessous de la ligne de menu, *Control+Clr/Home* le fait sauter au début du listing du programme.

La touche *Undo* permet d'annuler toutes les modifications qui ont été apportées à une ligne de programme, à condition que le curseur ne soit pas encore sorti de cette ligne. La touche *Help* permet de "replier" ou de "dérouler" les procédures dans le listing du programme.

De quoi s'agit-il ? Lorsque le curseur est placé sur une ligne où figure le mot d'instruction PROCEDURE et que vous appuyez alors sur la touche *Help*, toutes les lignes d'instructions jusqu'à la prochaine instruction RETURN (comprise) ne sont plus affichées dans le listing. Pour signaler cette situation, une flèche '>' est placée devant le mot d'instruction PROCEDURE.

Dans cette situation, aucune modification ne peut être apportée au nom de la procédure ou à la liste de paramètres du sous-programme. Pour rendre à nouveau visibles les lignes d'instructions entre PROCEDURE et RETURN, il suffit d'amener le curseur sur la flèche en début de ligne et d'appuyer à nouveau sur la touche *Help* ou bien d'effacer le caractère '>'.

Cette mémorisation (folding) d'un sous-programme permet de réaliser des listings courts et bien lisibles, dans lesquels on ne déroule que le sous-programme sur lequel on est en train de travailler. Voici par exemple comment pourrait se présenter un programme avec des procédures mémorisées :

```
init
main_menu
'
> PROCEDURE init
> PROCEDURE main_menu
> PROCEDURE ligne_menu
> PROCEDURE charger
```

```
> PROCEDURE sauvegarder
> PROCEDURE traiter
> PROCEDURE lire_infos
> PROCEDURE afficher_infos
```

Avec une procédure déroulée, le programme se présenterait alors ainsi :

```
init
main_menu
'
> PROCEDURE init
> PROCEDURE main_menu
> PROCEDURE ligne_menu
PROCEDURE charger
  FILESELECT "\*.RSC","",fichier_rsc$
  IF NOT EXIST(fichier_rsc$)
    ALERT 1,"Fichier n'existe pas !",1,"Arrêt",r%
  ELSE
    IF RSRC_LOAD(fichier_rsc$)=0
      ALERT 1,"Erreur lors du chargement du fichier !",1,"Arrêt",r%
      END
    ENDIF
  ENDIF
RETURN
> PROCEDURE sauvegarder
> PROCEDURE traiter
> PROCEDURE lire_infos
> PROCEDURE afficher_infos
```

LE BLOC DE TOUCHES NUMERIQUES

Le bloc de touches numériques sert normalement à l'entrée de chiffres et de quelques autres caractères. Il peut cependant également être appelé en liaison avec la touche *Control*. Les touches de ce bloc ont alors sensiblement la même fonction que sur les claviers avec NUMLOCK activé (par exemple sur les PC) :

Control et 4	Curseur un caractère vers la gauche
Control et 6	Curseur un caractère vers la droite
Control et 8	Curseur une ligne vers le haut
Control et 2	Curseur une ligne vers le bas
Control et 7	Sauter au début du programme
Control et 1	Sauter à la fin du programme
Control et 9	"Feuilleter" une page en arrière
Control et 3	"Feuilleter" une page en avant
Control et 0	Equivaut à Insert
Control et .	Equivaut à Delete

Le bloc de touches numériques peut aussi être basculé dans un mode sous lequel ces instructions pourront être appelées sans qu'il soit nécessaire d'actionner la touche *Control*. La commutation s'effectue à l'aide de la touche *Control* combinée à d'autres touches du bloc de touches numériques.

Control et - ainsi que *Control et (* équivalent à NUMLOCK. Après l'une de ces combinaisons, le fait d'appuyer sur une des touches du bloc numérique produira l'instruction présentée plus haut avec *Control*. Le mode NUMLOCK peut être abandonné en actionnant à nouveau les mêmes touches.

AUTRES TOUCHES D'EDITION SUR LE BLOC PRINCIPAL DU CLAVIER

La touche *Delete* permet d'effacer le caractère sur lequel figure le curseur ; le reste de la ligne est ramené en arrière. La touche *Backspace* efface le caractère placé à gauche du curseur et ramène également le reste de la ligne en arrière.

Le fait d'appuyer sur la touche *Tab* (tabulation) fait sauter le curseur à la prochaine position de tabulation. Ces positions sont placées tous les huit caractères. *Control+Tab* ramène le curseur d'une position de tabulation vers la gauche.

Les touches *Return* et *Enter* font sauter le curseur au début de la ligne suivante. En appuyant sur la touche *Escape,* on entre en *mode direct*.

INSTRUCTIONS CONTROL

Nous avons déjà signalé de nombreuses instructions *Control* dans les sections précédentes. Nous allons ici récapituler ces instructions ainsi que les instructions avec la touche *Control* que nous n'avons pas encore évoquées (à l'exception des instructions *Control* qui fonctionnent avec le bloc de touches numériques) :

Control+Delete
efface la ligne sur laquelle figure le curseur.

Control+U (undelete)
réinsère au contraire la dernière ligne à avoir été effacée avec *Control+Delete* ou *Control+Y*. Cela permet, d'une part, de rétablir les lignes qui ont été effacées par erreur mais aussi, d'autre part, de déplacer ou de copier très facilement des lignes isolées (pour dupliquer une ligne par exemple : tenir la touche *Control* enfoncée puis actionner *Delete* et deux fois U).

Control+Y
efface la ligne sur laquelle figure le curseur.

Control+N
insère une ligne vide au-dessus de la ligne d'instruction dans laquelle figure le curseur (comme *Insert*), même si des modifications ont été apportées à la ligne.

Control+Q
Appeler le menu de bloc (comme la touche de fonction F4).

Control+B
Marquer le début d'un bloc.

Control+K
Marquer la fin d'un bloc.

Control+R
"feuilleter" une page en arrière.

Control+C
"feuilleter" une page en avant.

Control+E
Remplacer texte.

Shift+Control+E
Remplacer avec demande de chaîne à rechercher et de chaîne de remplacement.

Control+F (find)
Chercher texte.

Shift+Control+F
Rechercher texte avec demande de chaîne à rechercher.

Control+Curseur gauche
Sauter au début de la ligne.

Control+Curseur droite
Sauter à la fin de la ligne.

Control+Curseur haut
"feuilleter" une page en arrière.

Control+Curseur bas
"feuilleter" une page en avant.

Control+Clr/Home
Sauter au début du programme.

Control+Z
Sauter à la fin du listing du programme.

Control+Tab
Sauter d'une position de tabulation vers la gauche.

Control+G (goto)
Ouvrir le champ d'affichage des numéros de lignes pour permettre l'entrée d'un numéro de ligne (auquel on sautera alors).

Un groupe particulier d'instructions *Control* permet de fixer des marques sous l'éditeur ou bien de sauter à ces marques. Ces marques ne valent que pour l'éditeur, elles n'ont donc rien à voir avec les marques utilisées par GOTO ou RESTORE. Ces marques d'édition peuvent être fixées dans l'emplacement du curseur en appuyant sur *Control* et sur un chiffre du clavier principal. On peut ensuite sauter à la marque voulue en appuyant sur *Alternate* et sur ce même chiffre.

Les combinaisons de touches *Alternate* avec les chiffres 7 à 9 et 0 sont prédéfinies. En appuyant sur *Alternate et 7,* on saute à la dernière position du curseur avant le passage en *mode direct* ou bien avant le dernier lancement du programme. *Alternate et 8* saute à l'emplacement où figurait le curseur lors du lancement de l'éditeur. Avec *Alternate et 0,* on saute à la position qu'occupait le curseur lors de la dernière modification effectuée. *Alternate et 9* saute à l'emplacement à partir duquel a été lancé la dernière opération de recherche.

LA BARRE DES MENUS ET LES TOUCHES DE FONCTION

Dans les deux lignes du haut de l'écran figurent, sous l'éditeur, deux lignes de menu. Tout à gauche figure le symbole Atari qui permet, lorsque vous le cliquez, de dérouler un menu *pull down.* Ce menu contient les titres de menu 'symbole Atari' et 'GFA-BASIC'.

Sous le symbole Atari apparaît, lorsque vous cliquez ce point du menu, GFA-BASIC 3.00 D, le numéro de version, avec deux possibilités de sélection, *Editor* et *Menu.* Si vous cliquez le bouton *Editor,* vous revenez à l'éditeur GFA-BASIC. Si vous sélectionnez par contre *Menu,* vous revenez au menu déroulant.

Si vous avez chargé des accessoires de bureau (par exemple CONTROL.ACC ou EMULATOR.ACC), vous aurez aussi la possibilité de les sélectionner à partir de ce menu.

Le titre de menu GFA-BASIC contient les points suivants :

Save
saute à l'éditeur, après quoi apparaît un sélecteur de fichier dans lequel vous pouvez indiquer un nom de fichier sous lequel le programme actuel peut être sauvegardé.

Load
saute à l'éditeur, après quoi apparaît un sélecteur de fichier dans lequel vous pouvez indiquer le nom du fichier qui doit être chargé.

Deflist
permet de fixer le mode *Deflist* voulu à l'aide d'un sélecteur d'objet (voyez la section *Particularités*).

Nouveaux noms
Ce point du menu permet de passer dans un mode sous lequel l'introduction de nouvelles variables dans le programme actuel devra être confirmée. Ce mode n'est pas activé au départ. Lorsqu'il est activé, chaque fois qu'une variable ou un nom de procédure ou de fonction qui n'a pas encore été utilisé est employé, une boîte d'alerte apparaît dans laquelle on doit indiquer si cette nouvelle variable doit bien être admise dans le programme ou bien s'il s'agit d'une faute de frappe (dans ce cas, le message d'erreur *"Syntax Error"* apparaîtra).

Sur le côté droit de la barre des menus figure en haut une montre sous laquelle est affiché le numéro de la ligne de texte sur laquelle se trouve le curseur. L'emploi de ces deux éléments de la ligne du menu sera expliqué à la fin de cette section.

Entre le symbole Atari et la montre figurent vingt mots d'instructions, par groupe de deux superposés, que vous pouvez sélectionner en cliquant avec la souris ou bien appeler à l'aide des touches de fonction. La ligne inférieure peut être appelée en actionnant simplement une touche de fonction, alors que les instructions de la ligne du haut peuvent être appelées en appuyant simultanément sur la touche *Shift* et sur la touche de fonction correspondante. Au mot en bas à gauche de la ligne de menu *(Load)* correspondra par exemple la touche F1, le mot d'instruction au-dessus *(Save)* étant appelé avec SHIFT+F1.

Sous le symbole Atari, un emplacement correspondant à deux autres caractères est laissé vide au départ. C'est là qu'est affiché l'état des touches CAPS-LOCK et NUM-LOCK. Si la touche CAPS-LOCK est enfoncée, une flèche vers le haut apparaît à gauche sous le symbole Atari. Si la touche NUM-LOCK est enfoncée, c'est-à-dire *Control+(*, le symbole exposant ^ apparaît à droite sous le symbole Atari. Les modes NUM-LOCK et CAPS-LOCK peuvent aussi être sélectionnés en cliquant avec le bouton gauche de la souris sous le symbole Atari.

Les points du menu ont la signification suivante :

Load (F1)

L'instruction *Load* permet de charger un programme GFA-BASIC 3.0. Le format utilisé ici emploie ce qu'on appelle des *tokens d'instructions*. Ce type de fichier de programme peut être chargé et sauvegardé beaucoup plus rapidement. La marque de fichier (l'extension) .GFA est normalement utilisée pour désigner ce format.

Les versions antérieures de GFA-BASIC emploient un autre format de tokens. Vos anciens programmes doivent donc tout d'abord être sauvegardés avec *Save,A* sous l'ancienne version, puis être chargés sous GFA-BASIC 3.0 avec *Merge*. Vous pouvez alors les sauvegarder en 3.0 avec *Save*, après quoi ils pourront aussi être chargés normalement sous GFA-BASIC 3.0 avec *Load*.

Save (Shift+F1)

Un sélecteur de fichier apparaît, dans lequel vous pouvez indiquer un nom. Vous pouvez sauvegarder sous ce nom le programme figurant actuellement sous l'éditeur. Le programme sera dans ce cas sauvegardé sous le format dit de tokens évoqué sous *Load*. La marque de fichier (extension) prédéfinie est GFA. Cela signifie que cette marque de fichier est automatiquement ajoutée au nom de fichier si l'utilisateur n'indique aucune extension.

S'il existe déjà un fichier portant le nom spécifié, cet ancien fichier sera sauvegardé sous le même nom de fichier avec l'extension .BAK.

Merge (F2)

Cette instruction permet d'insérer dans le programme actuel un fichier en format ASCII. L'insertion s'effectue à partir de la ligne où se trouve le curseur. La marque de fichier prédéfinie est LST. Les programmes des anciennes versions GFA-BASIC doivent être sauvegardés sous l'ancienne version avec *Save,A* et être rechargés sous GFA-BASIC 3.0 avec *Merge*.

==> Des lignes de programme commençant par cette marque (deux signes
==> "égal" et un signe "supérieur") seront produites lorsque sera mergé un
==> programme que l'interpréteur ne parvient pas à comprendre.

Save,A Shift+F2

Save,A permet de sauvegarder le programme actuel en format ASCII. Un fichier sauvegardé sous ce format, c'est-à-dire sous forme de texte, peut être ensuite rechargé avec *Merge*. La marque de fichier (extension) prédéfinie est LST. Cette marque de fichier est automatiquement ajoutée au nom de fichier si l'utilisateur n'indique aucune extension.

S'il existe déjà un fichier portant le nom spécifié, cet ancien fichier sera sauvegardé sous le même nom de fichier avec l'extension .BAK.

Llist (F3)

Cette instruction déclenche l'impression du programme actuellement placé sous l'éditeur. Le format de l'impression peut être défini à l'aide de ce qu'on appelle des instructions de point. Ces instructions doivent être placées dans le listing du programme, comme des instructions normales. En voici la liste (x représente un chiffre quelconque) :

.ll xx	- longueur de ligne maximale
.pl xx	- longueur de page maximale
	- effectuer un saut de page
	- conditional page
	- numérotation des lignes
	- marge gauche
.ff xxx	- chaîne de caractères Form-Feed (une chaîne de remplacement peut être spécifiée pour les imprimantes attendant d'autres valeurs pour Form Feed. La valeur prédéfinie est .ff\012)
.he en-tête	- Texte de la ligne d'en-tête
.fo bas	- Texte de la ligne de bas de page
.lr xx	- Marge gauche
.l-	- Cette instruction a pour effet que les lignes suivantes n'apparaîtront plus dans le listing. Le listing pourra être à nouveau activé par un .l+ apparaissant plus loin dans le programme.
.l+	- Voyez .l-
.n1 à .n9	- active la numérotation des lignes avec un à neuf chiffres.
.n0	- Désactive la numérotation des lignes.

Des symboles particuliers peuvent être insérés dans le texte des lignes d'en-tête et de bas de page :

\xxx	- caractère de code ASCII xxx
\d	- date
\t	- heure
#	- numéro de page

Pour pouvoir imprimer les caractères spéciaux # et \, il faut les faire précéder d'un autre caractère \. La combinaison de caractères \\ donnera donc un \ sur le papier, \# un #.

Quit (Shift+F3)

Permet de quitter l'interpréteur GFA-BASIC.

Block **(F4)**

Si ce point du menu est sélectionné alors qu'aucun bloc n'est encore marqué, le texte *"Block ???"* apparaît dans la ligne du menu pour indiquer que la sélection de l'instruction *Block* est sans intérêt dans cette situation. Si par contre un bloc est marqué, un menu vient s'inscrire dans la ligne du haut de l'écran. Les différents points de ce menu peuvent être cliqués avec la souris mais ils peuvent aussi être activés en appuyant sur une lettre d'instruction. Les points du menu sont (entre parenthèses figure la lettre permettant d'appeler chaque point du menu) :

Copy (C)	: copie le bloc dans la position actuelle du curseur. Le bloc reste marqué.
Move (M)	: déplace le bloc dans la position actuelle du curseur. Le marquage du bloc est annulé.
Write (W)	: sauvegarde le bloc sous forme d'un fichier ASCII.
List (L)	: imprime le bloc.
Start (S)	: saute au début du bloc.
End (E)	: saute à la fin du bloc.
^Del (Touche Control + D)	: efface le bloc.
Hide (H)	: annule le marquage du bloc.

Le fait d'appuyer sur une autre touche (ou sur un bouton de la souris) en dehors de cet éventail de choix fait disparaître le menu de bloc.

New **(Shift+F4)**

Efface le programme figurant actuellement dans l'éditeur.

BlkEnd **(F5)**

BlkEnd sert à marquer comme fin de bloc la ligne placée avant la ligne du curseur. Si la marque de début de bloc est placée avant cette ligne, le bloc apparaît alors marqué. Dans la version monochrome, il se détache sur un fond grisé. Dans la version couleur, le bloc marqué est affiché dans une couleur différente. Cette instruction peut également être déclenchée en appuyant sur *Control + K*.

BlkSta **(Shift+F5)**

BlkSta marque comme début de bloc la ligne sur laquelle se trouve le curseur.

Si une marque de fin de bloc a déjà été fixée après cette ligne, le bloc se détache sur un fond grisé. Cette instruction peut également être déclenchée en appuyant sur *Control + B*.

Find (F6)

Après avoir appelé l'instruction *Find*, vous pouvez indiquer le texte qu'il s'agit de rechercher. La recherche commence par la ligne sur laquelle figure le curseur. La recherche peut être ensuite relancée avec *Control+F* ou *Control+L*, sans qu'on vous demande cette fois d'entrer un texte à rechercher.

Si le texte à rechercher a été trouvé, le curseur apparaît au début de la ligne contenant le texte recherché ; si ce texte n'a pu être trouvé, le curseur apparaît alors à la fin du programme.

Lorsque l'instruction *Find* est appelée à nouveau, la chaîne recherchée telle qu'elle avait été entrée la dernière fois apparaît dans la ligne servant à entrer la ligne recherchée. Vous disposez, pour l'entrée du texte recherché, des possibilités d'édition suivantes :

Curseur gauche : Curseur va un caractère sur la gauche (à moins qu'il ne soit déjà à l'extrême gauche).

Curseur droite : Curseur va un caractère sur la droite (à moins qu'il ne soit à la fin de la chaîne recherchée).

Delete : Efface le caractère de la chaîne recherchée placé sous le curseur, le reste de la chaîne recherchée étant ramené de droite à gauche.

Backspace : Efface le caractère à gauche du curseur (à moins que le curseur ne soit déjà à l'extrême gauche) et ramène en arrière le reste de la chaîne recherchée.

Escape : Vide le champ d'entrée.

Return/Enter : Confirme le texte recherché entré et lance la recherche.

L'instruction *Find* peut également être appelée à l'aide de la combinaison de touches *Shift+Control+F* ou *Shift+Control+L*. Aucun texte ne peut être trouvé à l'intérieur des procédures "repliées" (voyez la touche *Help* dans la section consacrée au bloc de touches du curseur).

Replac (Shift+F6)

Cette instruction sert à remplacer un texte par un autre. Après qu'elle ait été appelée, on vous demande tout d'abord d'entrer le texte qu'il s'agit de remplacer.

On vous demande ensuite le texte par lequel il devra être remplacé. La recherche du texte à remplacer commence à partir de la position du curseur. Une fois que le texte recherché a été trouvé, le curseur saute au début de la ligne correspondante.

Le remplacement peut alors être effectué en actionnant la combinaison de touches *Control+E*. Si vous appuyez à nouveau sur cette combinaison de touches, le prochain emplacement du texte à remplacer sera recherché ou bien ce texte sera immédiatement remplacé s'il figure toujours sur la ligne actuelle du curseur.

L'instruction *Replac* peut également être appelée à l'aide de la combinaison de touches *Shift+Control+E*. Les possibilités d'édition dont vous disposez pour l'entrée du texte à rechercher et du texte de remplacement sont les mêmes que celles décrites pour l'instruction *Find*. Aucun texte ne peut être trouvé à l'intérieur des procédures "repliées" (voyez la touche *Help* dans la section consacrée au bloc de touches du curseur).

Pg Down (F7)

"Feuillette" le texte du programme d'une page écran en avant. Peut également être appelée avec *Control+C*.

Pg Up (Shift+F7)

"Feuillette" le texte du programme d'une page écran en arrière. Peut également être appelée avec *Control+R*.

Insert/Overwr (F8)

Commute entre les modes d'insertion et d'effacement.

Txt 16/Text 8 (Shift+F8)

Cette instruction n'est disponible que sous la résolution monochrome. En mode 16, les lettres sont affichées avec une hauteur de 16 points écran, de sorte que 23 lignes sont simultanément visibles sur l'écran. En mode 8, la hauteur des lettres n'est plus que de 8 points écran mais 48 lignes peuvent être affichées simultanément. Vous pouvez commuter entre ces deux modes avec *Txt 16/Text*.

Flip (F9)

Cette instruction permet de commuter sur l'image écran représentant la situation à laquelle a abouti le dernier programme appelé. Vous pouvez ensuite revenir à l'écran de l'éditeur en appuyant sur une touche (ou sur un bouton de la souris).

Direct **(Shift+F9)**

Cette instruction appelle le mode direct. Ce mode permet d'entrer des instructions GFA-BASIC qui sont exécutées dès que vous appuyez sur la touche *Return* ou *Enter*. Certaines instructions, telles que les instructions de boucle par exemple, ne peuvent toutefois être appelées en mode direct. Ce mode peut aussi être appelé en appuyant sur la touche *Escape*.

Le mode direct permet en outre de répéter les huit dernières instructions (qui avaient été entrées en mode direct) à l'aide des touches *Curseur haut* et *Curseur bas*.

En appuyant sur la touche *Undo*, on peut faire afficher la dernière instruction. Les combinaisons de touches *Control+Curseur gauche* ou *Control+Curseur droite* permettent de sauter respectivement au début et à la fin de la ligne. En appuyant sur la touche *Insert*, on peut en outre commuter entre les modes d'insertion et d'effacement.

Vous pouvez quitter le mode direct en appuyant sur la touche *Escape* suivie de *Return* ou bien en appuyant sur *Control+Shift+Alternate*.

Pour appeler des instructions de plusieurs lignes, vous pouvez écrire sous l'éditeur une procédure contenant ces instructions. Cette procédure pourra alors être appelée en mode direct.

Test **(F10)**

Lorsque cette instruction est appelée, GFA-BASIC teste si toutes les boucles, tous les sous-programmes et toutes les instructions conditionnelles du programme actuel ont été correctement refermées. C'est donc un examen de la structure du programme qui est effectué.

Run **(Shift+F10)**

Lance le programme figurant actuellement sous l'éditeur. Si ce programme comporte une erreur de structure, par exemple une boucle FOR-NEXT non refermée, un message d'erreur approprié est affiché et le programme n'est pas lancé.

Control+Shift+Alternate

La combinaison des touches *Control*, *Shift gauche* et *Alternate* permet d'interrompre un programme en cours d'exécution.

L'AFFICHAGE DE LIGNES ET LA MONTRE

En bas à droite de la ligne de menu est affiché un numéro qui indique sur quelle ligne figure actuellement le curseur. C'est dans la même zone que vous pouvez aussi entrer un numéro de ligne pour faire sauter le curseur à la ligne correspondante. A cet effet, vous devez cliquer le champ d'affichage du numéro de ligne avec la souris ou bien employer l'instruction *Control+G*.

Dans ce champ, seuls les chiffres sont acceptés comme entrée. Les possibilités d'édition pour l'entrée du numéro de ligne voulu sont les suivantes :

Curseur gauche : Curseur va un caractère sur la gauche (à moins qu'il ne soit déjà à l'extrême gauche).

Curseur droite : Curseur va un caractère sur la droite (à moins qu'il ne soit à la fin du champ d'entrée).

Backspace : Même fonction que curseur gauche.

Escape : Vide le champ d'entrée.

Return/Enter : Confirme le nombre entré.

Au-dessus de l'affichage du numéro de ligne figure une montre qui affiche l'heure système. En cliquant la montre avec la flèche de la souris, vous avez la possibilité de fixer à nouveau l'heure. Les possibilités d'édition pour l'entrée de l'heure sont les mêmes que pour l'entrée dans le champ du numéro de ligne. Seule la touche *Escape* a une autre signification, à savoir qu'elle sert ici à interrompre l'entrée, l'ancienne valeur de l'heure étant alors conservée.

PARTICULARITES

DEFBIT f$
DEFBYT f$
DEFINT f$
DEFWRD f$
DEFFLT f$
DEFSTR f$

```
f$ : Constante de chaine
```

L'instruction DEFxxx permet une déclaration simplifiée des variables. xxx symbolise ici une abréviation indéterminée de type de variable parmi celles présentées et expliquées dans la liste ci-dessous :

DEFxxx	f$	Suffixe	Toutes les variables
DEFBIT	"b"	!	commençant par 'b' sont déclarées comme des variables booléennes.
DEFBYT	"by"	|	commençant par les deux lettres 'by' sont déclarées comme variables entières sur un octet.
DEFWRD	"w"	&	commençant par la lettre 'w' sont déclarées comme variables entières sur 2 octets avec signe.
DEFINT	"i-k,m-p"	%	commençant par les lettres 'i' à 'k' et 'm' à 'p' sont déclarées comme variables entières sur 4 octets avec signe.
DEFFLT	"x-z"	#	commençant par les lettres 'x' à 'z' sont déclarées comme valeurs à virgule flottante sur 8 octets.
DEFSTR	"s,t"	$	commençant par les lettres 's' et 't' sont déclarées comme chaines de caractères.

Les spécifications DEFSNG ou DEFDBL sont automatiquement remplacées par l'éditeur par DEFFLT.

Il est en principe préférable d'effectuer ces déclarations globales de variables au début du programme. Pour utiliser, dans n'importe quel endroit du programme, une variable ne se conformant pas à la définition fixée avec DEFxxx, il suffit d'indiquer le suffixe voulu à la suite du nom de la variable. Cette spécification explicite du type de variable a toujours priorité sur les définitions globales.

La déclaration des types de variables vaut jusqu'à ce qu'une autre déclaration soit effectuée. Le type de variable prédéfini est celui de valeur numérique à virgule flottante sur 8 octets.

L'affichage des types de variables ainsi que l'écriture en majuscules ou minuscules dans le listing est défini par l'instruction DEFLIST.

DEFLIST n

```
(n : iexp)
```

DEFLIST définit le format du listing du programme. L'expression numérique n peut recevoir une valeur entre 0 et 3 (compris). La table suivante décrit l'effet de l'instruction DEFLIST sur le mode d'écriture des noms d'instructions et de variables :

n	Instruction	Variable
0	PRINT	abc
1	Print	Abc
2	PRINT	abc#
3	Print	Abc#

Le mode prédéfini est DEFLIST 0.

DEFLIST 0

Les instructions et fonctions GFA-BASIC sont affichées en majuscules. Les noms de variables, de procédures et de fonctions sont écrits en minuscules.

DEFLIST 1

Les instructions et fonctions GFA-BASIC ainsi que les noms de variables, de procédures et de fonctions sont écrits avec l'initiale en majuscule, le reste du nom étant écrit en minuscules.

DEFLIST 2

Comme DEFLIST 0, si ce n'est qu'une marque (un suffixe) sera en outre ajouté à tous les noms de variables.

DEFLIST 3

Comme DEFLIST 1, si ce n'est qu'une marque (un suffixe) sera en outre ajouté à tous les noms de variables.

$ texte

```
texte : Séquence de caractères quelconques
```

L'instruction $ qui est traitée par l'interpréteur comme un REM sert à la commande du compilateur. Vous en trouverez une description détaillée dans le manuel d'utilisation de la version du compilateur correspondant à GFA-BASIC 3.0.

2. VARIABLES ET GESTION DE LA MEMOIRE

TYPES DE VARIABLES

GFA-BASIC 3.0 dispose des types de variables suivants :

Nom	Suffixe	Place mémoire occupée
Boolean	!	1 octet (un bit dans les tableaux)
Byte	|	1 octet
Word	&	2 octets
Integer	%	4 octets
Float	#	8 octets
String	$	suivant la longueur de la chaîne

Les variables booléennes ne peuvent revêtir que les deux valeurs 0 (FALSE) ou -1 (TRUE). Si une valeur différente de zéro leur est affectée, cette valeur est interprétée comme étant -1. Ce type de variable est doté du suffixe ! et il occupe un octet de place mémoire. Dans les tableaux booléens, un élément de tableau occupe seulement un bit.

Exemples :
```
b!=TRUE
c!=x>y
```

Le type de variable *Byte* peut recevoir des valeurs comprises entre 0 et 255. Les valeurs plus élevées pour x sont traitées comme un débordement. Le suffixe de ce type de variable est le trait vertical. Comme l'indique son nom, ce type de variable occupe un octet (= byte en anglais).

Exemple :
```
x|=128
```

Word est un type entier sur 2 octets avec signe. Le suffixe de ce type est &. Le domaine numérique qu'il permet d'afficher va de -32768 à 32767.

Exemple :
```
x&=32767
```

Integer est un type entier sur 4 octets avec signe. Le signe % est utilisé comme suffixe. Le domaine numérique couvert va de -2147483648 à 2147483647.

Exemple : `x%=2000000000`

Float est un type de variables à virgule flottante sur 8 octets. Ce type est désigné par l'absence de suffixe ou bien au contraire par le signe #. Le domaine numérique couvert va de 2.225073858507E -308 à 3.595386269725E +308.

Les chaînes de caractères (*strings* en anglais) sont désignées par le suffixe $. Elles peuvent atteindre une longueur maximale de 32767 caractères. Les chaînes sont gérées à l'aide de ce qu'on appelle un descripteur. Un descripteur a une longueur de 6 octets. Les quatre premiers octets contiennent l'adresse de la chaîne de caractères, les deux derniers la longueur de la chaîne. Un octet de remplissage est placé à la suite de la chaîne de caractères si sa longueur est impaire, après quoi figure l'adresse du descripteur (*Backtrailer*).

Les adresses de tous les types de variables peuvent être déterminées à l'aide des fonctions VARPTR (ou V:) et ARRPTR (ou *). Pour les chaînes, VARPTR fournit l'adresse du premier octet de la chaîne de caractères elle-même et ARRPTR (ou *) l'adresse du descripteur de la chaîne.

Pour les tableaux, VARPTR / V: permet de déterminer les adresses des différents éléments (par exemple V: x%(5)). ARRPTR /* permet d'obtenir l'adresse du descripteur de tableau (par exemple ARRPTR (x%())) et VARPTR / V: détermine l'adresse des différents éléments du tableau.

TABLEAUX

DIM, DIM?
OPTION BASE
ARRAYFILL

Des tableaux peuvent être mis en place à partir de tous les types de variables. La mise en place d'un tableau s'effectue à l'aide de l'instruction DIM. La fonction de test DIM? permet de savoir combien d'éléments compte un tableau.

Les tableaux sont gérés dans la mémoire à l'aide de descripteurs. Un descripteur est une structure de six octets de long dont les quatre premiers octets contiennent l'adresse du tableau. Les deux octets suivants indiquent le nombre de dimensions du tableau. Le tableau commence par quatre octets pour chaque dimension du tableau, en commençant par la dernière dimension. Vient ensuite le contenu des différents éléments du tableau. Pour les variables alphanumériques, il s'agit en fait des descripteurs des chaînes de caractères.

Exemple : `DIM a%(2,3)`

entraînera la situation suivante : *a%() fournira l'adresse du descripteur de tableau.

Le nombre de dimensions du tableau sera fourni par les deux derniers des six octets constituant le descripteur de tableau. Donc :

```
INT{*a%()+4}
```

donnera la valeur 2. Le tableau commence par le nombre de subdivisions de la seconde dimension, l'élément zéro étant également pris en compte (si OPTION BASE 0 s'applique). Vient ensuite le nombre de subdivisions de la première dimension. Donc :

```
{{*a%()}}        donnera la valeur 4 et
{{*a%()}+4}      la valeur 3.
```

Viennent ensuite les 12 éléments du tableau, chacun occupant 4 octets dans la mémoire, dans l'ordre suivant :

```
a%(0,0)  a%(1,0)  a%(2,0)  a%(0,1)  a%(1,1)  a%(2,1)  etc...
```

DIM x(d1, [d2,...]) [,y(d1, [d2,...])]
DIM?(x())

```
(x,y : nom de variable (type de variable quelconque))
(d1,d2 : iexp)
```

L'instruction DIM permet de mettre en place des tableaux numériques ou des tableaux de chaînes de caractères (des tableaux alphanumériques). La structure de ces tableaux a été décrite dans l'introduction à cette section.

Le nombre de dimensions que peut comporter un tableau est illimité dans la pratique. Le nombre d'éléments des tableaux à plusieurs dimensions est limité en ce sens que les première et dernière dimensions ne doivent pas comporter plus de 65535 éléments. Le produit des nombres d'éléments des différentes dimensions du tableau doit de même être inférieur à 65535. (DIM a%(100,10,10) sera par exemple admis puisque la dernière dimension (10) et le produit des nombres d'éléments des différentes dimensions du tableau (100*10*10 = 10000) sont inférieurs à 65535).

OPTION BASE permet de définir si les tableaux comportent un élément numéro zéro.

Dans un tableau ne peuvent cohabiter que des variables du même type. Les éléments d'un tableau sont appelés par leurs indices. La fonction DIM? permet de déterminer combien d'éléments compte un tableau.

Exemple :

```
DIM x(10)
x(4)=3
PRINT x(LEN("Test"))
PRINT DIM?(x())
DIM y%(2,3)
PRINT DIM?(y%())
```

--> mettra en place deux tableaux. Une valeur sera écrite dans l'un des tableaux et immédiatement relue. Le nombre d'éléments des deux tableaux sera écrit sur l'écran (11 et 12).

OPTION BASE 0
OPTION BASE 1

L'instruction OPTION BASE permet de déterminer si les tableaux doivent ou non comprendre un élément numéro zéro. OPTION BASE 0 prévoit cet élément numéro zéro, alors que OPTION BASE 1 l'exclut. Dans ce cas, les indices de tous les tableaux commencent par 1.

Le contenu des tableaux n'est pas modifié par l'instruction OPTION BASE mais l'indice des différents éléments peut être modifié de 1.

Si aucune instruction OPTION BASE n'est employée, les tableaux comprennent des éléments d'indice 0.

Exemple :

```
DIM x%(3)
FOR i%=3 DOWNTO 0
  x%(i%)=i%
  PRINT i%,x%(i%)
NEXT i%
OPTION BASE 1
FOR i%=3 DOWNTO 0
  PRINT i%,x%(i%)
NEXT i%
```

--> écrit les indices et le contenu du tableau x%() sur l'écran. Le programme se termine par le message d'erreur *'Index de champ trop grand'* car x%() ne comporte plus d'élément numéro zéro à la suite d'OPTION BASE 1.

ARRAYFILL x(),y

```
x : Nom d'un tableau avec type de variable numérique
y : aexp
```

L'instruction ARRAYFILL fixe tous les éléments du tableau x sur une valeur égale à celle de l'expression numérique y.

Exemple :

```
DIM x(10)
PRINT x(4)
ARRAYFILL x(),5+1
PRINT x(4)
```

--> affiche sur l'écran le nombre zéro car l'instruction DIM fixe automatiquement tous les éléments du tableau sur zéro. Après remplissage du tableau, le nombre 6 est sorti sur l'écran.

CONVERSION DE TYPE

TYPE(x)

```
(x : iexp)
```

La fonction TYPE permet de déterminer si un pointeur correct a été transmis pour une variable. Un code définissant le type de cette variable ou bien la valeur -1 est renvoyé. Suivant le type de variable désigné par x, on obtient les valeurs suivantes :

Float (virgule flottante)	-->	0
String (chaîne)	-->	1
Integer (entière)	-->	2
Boolean (booléenne)	-->	3
Float Array (tableau à virgule flottante)	-->	4
String Array (tableau de chaînes)	-->	5
Integer Array (tableau entier)	-->	6
Boolean Array (tableau booléen)	-->	7
Word (mot)	-->	8
Byte (octet)	-->	9
Word Array (tableau de mots)	-->	12
Byte Array (tableau d'octets)	-->	13

Exemple :

```
a$="test"
x%=4
DIM y(3)
PRINT TYPE(*a$),TYPE(*y())
```

--> écrit les nombres 1 et 4 sur l'écran.

ASC(a$)
CHR$(x)

```
(a$ : sexp)
(x : aexp)
```

ASC et CHR$ sont deux fonctions inverses.

La fonction ASC fournit le code ASCII du premier caractère de la chaîne de caractères a$. Si a$ est une chaîne vide, un zéro est fourni en réponse.

CHR$ fournit le caractère ASCII portant le code x. Seul l'octet de plus faible poids de x est évalué (ce qui correspond à l'expression x AND 255).

Exemple :

```
PRINT ASC("TEST")
code|=ASC(CHR$(65))  ! CHR$(65) donnera 'A'
PRINT code|,CHR$(189)
```

--> écrit les nombres 84, 65 et le symbole du Copyright sur l'écran.

Les fonctions STR$, BIN$, OCT$ et HEX$ convertissent une expression numérique x en une chaîne de caractères contenant le nombre calculé à partir de l'expression numérique x.

y définit la longueur qui devra être celle de la chaîne de caractères renvoyée. Si *y* est supérieur au nombre de caractères nécessaires pour représenter le nombre voulu, la chaîne sera comblée au début par des espaces pour STR$. Avec les fonctions BIN$, OCT$ et HEX$, la chaîne de résultat sera comblée avec des zéros si nécessaire.

STR$(x)
STR$(x,y)
STR$(x,y,z)

```
(x,y,z : aexp)
```

STR$ convertit un nombre *'x'* en une chaîne de caractères. Un second paramètre *'y'* permet de spécifier sur combien de positions la chaîne-résultat doit être remplie d'espaces sur la gauche ou bien combien de caractères de la chaîne-résultat (en comptant à partir de la fin de cette chaîne) doivent être employés.

STR$(x,n) équivaut à RIGHT$(SPACE$(n)+STR$(x),n).

Une autre variante de STR$ dispose d'un troisième paramètre *'z'*. Dans ce cas, le nombre *'x'* est formaté et arrondi avec *'y'* positions dont *'z'* chiffres après la virgule.

Exemple :

```
a=123.4567
PRINT STR$(a,6,2)
PRINT STR$(PI,5,3)
PRINT STR$(PI,2,2)
```

--> écrit les nombres 123.46, 3.142 et 14 sur l'écran.

BIN$(x [,y])
OCT$(x [,y])
HEX$(x,y)

```
(x,y : iexp)
```

BIN$ convertit un nombre entier en format binaire. Cela signifie que le nombre apparaîtra, après conversion, sous forme de son équivalent dans le système numérique de base 2. Ce système numérique représente tous les nombres à l'aide des seuls chiffres 0 et 1. Le paramètre *'y'* indique combien de chiffres doivent être utilisés (de 1 à 32).

OCT$ convertit une valeur entière en son équivalent octal, c'est-à-dire en son équivalent dans le système numérique de base 8. Ce système numérique représente tous les nombres à l'aide des chiffres 0 à 7. Le paramètre optionnel *'y'* indique combien de chiffres doivent être utilisés (de 1 à 11).

HEX$ convertit un nombre entier en son équivalent dans le système hexadécimal. Ce système numérique est de base 16 et il emploie les chiffres 0 à 9 et les lettres A à F. Le second paramètre 'y', optionnel, sert à indiquer combien de chiffres hexadécimaux doivent être affichés (huit au maximum).

HEX$(x,n) équivaut à RIGHT$(STRING$(n,48)+HEX$(x),n).

Exemples :

```
x=32+15
a$=OCT$(16+7,4)
PRINT HEX$(x),a$,BIN$(1+4+16+64,8)
```

--> écrit 2F,0027 et 01010101 sur l'écran.

VAL(a$)
VAL?(a$)

```
(a$ : sexp)
```

VAL() convertit une chaîne de caractères en un nombre. Si VAL() rencontre un caractère qui ne peut être interprété comme partie d'un nombre, la conversion de la chaîne est interrompue. C'est alors le nombre inscrit au début de la chaîne qui est renvoyé par VAL. Si la chaîne ne commence pas par un nombre, VAL renvoie la valeur zéro.

En spécifiant &H(hex), &X(bin) ou &O(oct), vous pouvez faire convertir des nombres exprimés en système binaire, octal ou hexadécimal.

$ permet en outre de désigner des nombres hexadécimaux et % des nombres binaires.

VAL?() permet de tester combien de chiffres d'un nombre seraient convertis avec VAL(). VAL?() fournit zéro comme résultat si aucun nombre valable ne peut être lu.

Exemples :

```
a$=STR$(12345)
PRINT VAL(a$),VAL("-.123 abc 123"),VAL?("3.00 FF")
```

--> affichera sur l'écran les nombres 12345, -.123 et 4.

```
PRINT VAL("&H"+"AF")
```

--> affiche le nombre 175.

```
PRINT VAL("$AA")
PRINT VAL("%10101010")
```

--> affiche deux fois le nombre 170.

CVI(a$) CVL(a$) CVS(a$) CVF(a$) CVD(a$) MKI$(x) MKL(x) MKS(x) MKF(x) MKD(x)

```
(a$ : sexp)
(x : aexp)
```

Les fonctions CVI, CVL, CVS, CVF et CVD convertissent des chaînes de caractères en nombres. Contrairement à VAL/STR$, aucune conversion en clair n'est effectuée mais c'est la représentation interne de la chaîne et des nombres qui entre en ligne de compte. Voici les effets des différentes fonctions CVx :

CVI convertit une chaîne de caractères de 2 octets en un nombre entier de 16 bits.

CVL convertit une chaîne de caractères de 4 octets en un nombre entier de 32 bits.

CVS convertit une chaîne de caractères de 4 octets contenant un nombre en format compatible avec le BASIC ST en un nombre à virgule flottante GFA-BASIC.

CVF convertit une chaîne de caractères de 6 octets en format GFA-BASIC 1.0 et en format GFA-BASIC 2.0 en un nombre à virgule flottante (en format GFA-BASIC 3.0).

CVD convertit une chaîne de caractères de 8 octets en un nombre à virgule flottante en format GFA-BASIC 3.0.

MKI$, MKL$, MKS$, MKF$ et MKD$ sont les fonctions inverses des fonctions CVx. Leur rôle peut donc également être déduit de la table ci-dessus mais il est précisé par les équivalences suivantes :

```
MKI$(x%)=CHR$(SHR(x%,8))+CHR$(x%)
MKL$(x%)=CHR$(SHR(x%,24))+CHR$(SHR(x%,16))+CHR$(SHR(x%,8))+CHR$(x%)
```

Il convient donc de noter que l'octet de poids fort passe en premier.

La lecture des formats numériques de programmes externes ainsi qu'un stockage des nombres compactés, par exemple dans des fichiers "R", constituent des exemples d'applications caractéristiques de ces fonctions.

Exemple :

```
a$=MKL$(1000)
PRINT CVL(a$),LEN(a$)
b$=MKD$(100.1)
PRINT CVD(b$),LEN(b$)
```

--> écrit sur l'écran les nombres 1000 et 4 ainsi que 100.1 et 8.

CINT(x)
CFLOAT(y)

```
(x : aexp)
(y : iexp)
```

La fonction CINT convertit un nombre à virgule flottante *'x'* en une valeur entière arrondie. CFLOAT convertit de même une valeur entière *'y'* en un nombre à virgule flottante. Cette fonction n'est pas utilisée normalement et nous ne la citons que par souci d'exhaustivité. Elle peut d'ailleurs jouer un certain rôle sous le compilateur.

Exemple :

```
a=1.2345
a%=10000
b%=CINT(a)
b=CFLOAT(a%)
PRINT b%,b
```

--> affiche 1 et 10000 sur l'écran.

OPERATIONS DE POINTEUR

(*)
(BYTE{}, CARD{}, INT{}, LONG{}, {}, FLOAT{},
SINGLE{}, DOUBLE{}, CHAR{})
(xPEEK, xPOKE)
(V:, VARPTR)
(ARRPTR)
(ABSOLUTE)

***x**

```
(x : svar ou un nom de tableau suivi de () )
```

Le signe de multiplication peut aussi faire office de symbole de pointeur. Dans ce cas, *x fournit l'adresse à laquelle figure la variable x dans la mémoire. Pour les chaînes de caractères, *x$ fournit l'adresse du descripteur de chaîne (ARRPTR(x$)). L'expression *x équivaut donc à ARRPTR(x). Cela vaut également pour les tableaux.

Si le symbole de pointeur est utilisé pour des tableaux, * étant simplement suivi du nom du tableau et de (), *x() équivaut à ARRPTR(x()). Cette variante de l'emploi des pointeurs joue surtout un rôle pour la transmission indirecte de tableaux à des sous-programmes. Sous la version 3.0, il est cependant encore préférable d'utiliser à cet effet l'instruction VAR.

Exemple :

```
' Transmission indirecte de tableau
DIM a(3)
change(*a())
PRINT a(2)
'
PROCEDURE change(ptr%)
  SWAP *ptr%,x()
  ARRAYFILL x(),1
  SWAP *ptr%,x()
RETURN
```

--> le contenu du tableau a() est modifié sans que son nom apparaisse dans la procédure change. Le nombre 1 est écrit sur l'écran (voyez aussi SWAP).

Ou plutôt, en 3.0 :

```
DIM a(3)
change(a())
PRINT a(2)
'
PROCEDURE change(VAR x())
  ARRAYFILL x(),1
RETURN
```

PEEK(x) **DPEEK**(x) **LPEEK**(x)
POKE x,y **DPOKE** x,y **LPOKE** x,y
SPOKE x,y **SDPOKE** x,y **SLPOKE** x,y

```
(x,y : iexp)
```

La fonction PEEK, l'instruction POKE et leurs variantes permettent de lire des cellules de mémoire ou bien au contraire d'écrire des valeurs dans la mémoire. Les différentes variantes de cette instruction sont :

```
PEEK(x)   : lit un octet à l'adresse x
DPEEK(x)  : lit deux octets à partir de l'adresse x
LPEEK(x)  : lit quatre octets à partir de l'adresse x

POKE x,y  : écrit la valeur y à l'adresse x sous forme d'un octet
DPOKE x,y : écrit la valeur y à l'adresse x sous forme d'une valeur sur 2 octets
LPOKE x,y : écrit la valeur y à l'adresse x sous forme d'une valeur sur 4 octets
```

Lorsque vous employez DPEEK, LPEEK, DPOKE et LPEEK, vous devez veiller à n'indiquer que des adresses paires.

Des variantes de l'instruction POKE peuvent également fonctionner en mode *Supervisor* de sorte que même les adresses protégées, les adresses inférieures à 2048 peuvent être atteintes. Les instructions correspondantes s'appellent alors SPOKE, SDPOKE et SLPOKE. Il convient d'être particulièrement prudent en mode *Supervisor* car les manipulations des adresses protégées peuvent avoir des conséquences imprévisibles. Les fonctions *Peek* travaillent toutes en mode *Supervisor*.

Exemple :

```
LPOKE XBIOS(14,1)+6,0
```

--> vide le buffer du clavier (fixe les pointeurs *Head* (début) et *Tail* (fin) sur le début du buffer.

Ou bien, autre possibilité équivalente :

```
REPEAT
UNTIL INKEY$=""
```

--> vide le buffer clavier par une lecture caractère par caractère.

BYTE{x}
CARD{x}
INT{X}
LONG{x}
{x}
FLOAT{x}
SINGLE{x}
DOUBLE{x}
CHAR{x}

```
(x : iexp)
```

Ces instructions permettent de lire des types de variables déterminés à partir d'une adresse donnée ou bien d'en écrire à une adresse donnée. En tant que fonctions (par exemple y=BYTE{x}), elles permettent de lire à partir de l'adresse x de la mémoire. En tant qu'instructions proprement dites (par exemple BYTE{x}=y), elles écrivent la valeur y dans la mémoire à partir de l'adresse x.

Lorsque vous employez INT{}, CARD{}, WORD{}, LONG{}, FLOAT{}, SINGLE{} et DOUBLE{}, vous devez veiller à n'employer que des adresses paires, faute de quoi une erreur d'adresse (trois bombes) serait déclenchée.

Type		Rôle
BYTE(x)	:	lit/écrit un octet
CARD(x)	:	lit/écrit un entier sur 2 octets sans signe
INT(x)	:	lit/écrit un entier avec signe sur 2 octets
LONG(x)	:	lit/écrit une valeur entière sur quatre octets
(x)	:	lit/écrit une valeur entière sur quatre octets
FLOAT(x)	:	lit/écrit une variable à virgule flottante sur 8 octets, en format GFA-BASIC 3.0
SINGLE(x)	:	lit/écrit une variable à virgule flottante sur 4 octets en format single IEEE
DOUBLE(x)	:	lit/écrit une variable à virgule flottante sur 8 octets en format double IEEE
CHAR(x)	:	lit/écrit une chaîne terminée par un octet nul. Très important pour la communication avec les routines C, GEM et GEMDOS.

La ligne suivante :

```
x%=LONG(adr%)
```

affecte à la variable x% la valeur de long mot figurant à l'adresse adr% alors que

```
LONG(adr%)=x%
```

écrit la valeur de la variable x% à l'adresse adr% sous forme d'un long mot.

Les instructions SINGLE et DOUBLE permettent de lire ou d'écrire sous un format numérique étranger. Certains compilateurs C emploient par exemple le format DOUBLE IEEE. Pour convertir un nombre GFA-BASIC en format SINGLE ou DOUBLE et pour faire afficher le résultat en hexadécimal à titre de contrôle, on pourra écrire :

```
a$=SPACE$(4)
SINGLE(V:a$)=1.2345
PRINT HEX$(CVL(a$),8)
```

ou

```
a$=SPACE$(8)
DOUBLE(V:a$)=1.2345
PRINT HEX$((V:a$),8)
PRINT HEX$((V:a$+4),8)
```

Certaines des fonctions indiquées ci-dessus correspondent à la fonction PEEK et à ses variantes. LONG{x} (ou {x}) équivaut ainsi pratiquement à LPEEK(x) mais ne travaille pas en mode *Supervisor*. C'est pourquoi {x} est exécutée plus vite que LPEEK. Si vous tentez d'accéder avec cette instruction aux zones protégées de la mémoire (0 à 2047), une erreur de bus (deux bombes) sera provoquée pour signaler cette erreur de programmation.

Exemples :

```
adr%=XBIOS(2)
t%=TIMER
FOR i%=1 TO 4000
  ~LPEEK(adr%)
NEXT i%
PRINT (TIMER-t%)/200
t%=TIMER
FOR i%=1 TO 4000
  ~{adr%}
NEXT i%
```

```
PRINT (TIMER-t%)/200
'
FLOAT(*x)=PI
```

--> La première partie de l'exemple montre que {} travaille plus vite que LPEEK, la seconde partie écrit un nombre à virgule flottante dans la variable x de façon indirecte.

```
BYTE{XBIOS(2)+4160}=&HFF
CARD{XBIOS(2)+4320}=&HFFFF
LONG{XBIOS(2)+4480}=&HFFFFFFFF
'
a$="Mot de "+CHR$(0)
PRINT CHAR{V:a$};
b$=SPACE$(5)
CHAR{V:b$}="test"
PRINT b$,ASC(RIGHT$(b$))
```

--> Quelques valeurs sont tout d'abord écrites directement dans la mémoire écran où elles apparaissent sous forme de traits. Ensuite sont affectées à a$ et à b$ des chaînes de caractères terminées par un octet nul, l'affectation à a$ se faisant de la façon usuelle (avec +CHR$()) mais la sortie de façon inhabituelle avec CHAR{}. L'affectation à b$ se fait à travers CHAR{} et la sortie avec PRINT utilise le CHR$(0) qui n'est pas visible.

VARPTR(x) V:x
ARRPTR(y) *y

```
x : Nom d'une variable de type quelconque
y : Nom d'une variable ou d'un tableau avec parenthèses vides
```

ARRPTR(y) ou *y détermine l'adresse d'une variable y ou l'adresse du descripteur pour les tableaux ou chaînes.

VARPTR(x) ou V:x détermine par contre l'adresse de la chaîne de caractères elle-même ou, pour les tableaux, l'adresse d'un élément du tableau.

Pour les variables non scalaires (c'est-à-dire ni tableau ni chaîne), VARPTR, V:, ARRPTR et * reviennent au même.

Exemple :

```
DIM x%(10)
a$="Test"
PRINT ARRPTR(x%()),VARPTR(x%(0)),V:x%(1)
PRINT ARRPTR(a$),*a$,VARPTR(a$)
```

--> La première ligne sort l'adresse du descripteur de tableau ainsi que l'adresse des premiers éléments du tableau x%(). La seconde ligne sort deux fois sur l'écran l'adresse du descripteur de chaîne de a$ et une fois l'adresse du premier octet de la chaîne de caractères.

ABSOLUTE x,y

```
x : une variable de type quelconque
y : iexp
```

L'instruction ABSOLUTE modifie l'adresse (ARRPTR/*) d'une variable.

Exemple :

```
ABSOLUTE y,*x
x=13
y=7
PRINT x,y,*x,*y
```

--> La variable y est détournée sur l'adresse de x de sorte que finalement ces deux variables seront sorties avec la même valeur (7) et la même adresse.

SUPPRESSION ET ECHANGE

CLEAR, CLR, ERASE
SWAP
SSORT, QSORT
INSERT, DELETE

CLEAR
CLR x [,y,...]
ERASE z1() [,z2(),...]

```
x,y : svar ou avar
z1,z2 : Noms de tableaux quelconques
```

L'instruction CLEAR permet de supprimer toutes les variables et tous les tableaux. Cette instruction ne doit pas être utilisée à l'intérieur de boucles FOR-NEXT ou de sous-programmes. Elle est automatiquement exécutée lors du lancement d'un programme et ne présente d'intérêt pratique que lors du traitement des erreurs graves avec RESUME x.

L'instruction CLR supprime les variables énumérées dans la liste placée à la suite de l'instruction. CLR ne permet toutefois pas de supprimer des tableaux. ERASE permet de supprimer entièrement un tableau, qui peut alors être à nouveau dimensionné à la suite de cette instruction.

Contrairement à ce qui était le cas sous la version 2.0, il est également possible de supprimer plusieurs tableaux avec une seule instruction ERASE, par exemple ERASE x(),y().

Exemple :

```
x=2
y=3
CLEAR
PRINT x,y
'
x=2
y=3
CLR x
PRINT x,y
'
DIM x(10)
PRINT FRE(0)
ERASE x()
PRINT FRE(0)
```

--> écrit trois fois le nombre 0 et une fois le nombre 3 sur l'écran. Viennent ensuite deux nombres qui montrent que la place mémoire occupée par DIM peut à nouveau être libérée avec ERASE.

SWAP a,b
SWAP e(),f()
SWAP *c,d()

```
a,b   : avar ou svar
*c    : Pointeur sur un descripteur de tableau
d,e,f : Nom d'un tableau
```

Dans sa variante la plus simple, l'instruction SWAP sert à échanger les contenus de deux variables de même type (SWAP a,b). Elle peut cependant également être utilisée pour échanger deux tableaux. L'échange des tableaux se fait très rapidement puisque seuls les descripteurs correspondants sont en réalité échangés. Les dimensionnements des deux tableaux sont donc également échangés. Il n'est pas nécessaire pour cela que les tableaux soient déjà dimensionnés.

Dans la troisième variante, c contient le pointeur sur un descripteur de tableau et d() est le nom d'un tableau. Cela est particulièrement intéressant pour la transmission indirecte de tableaux aux sous-programmes (voyez le second exemple).

L'instruction SWAP ne doit pas être confondue avec la fonction SWAP qui est décrite dans la section consacrée aux opérations de bits.

Exemple :

```
x=1
y=2
PRINT x,y
SWAP x,y
PRINT x,y
```

--> affiche sur l'écran les nombres 1 et 2 puis 2 et 1.

```
DIM x(3)
change(*x())
PRINT x(2)
'
PROCEDURE change(adr%)
  SWAP *adr%,a()
  ARRAYFILL a(),1
  SWAP *adr%,a()
RETURN
```

--> Le tableau x() sera rempli de uns dans la procédure change, sans que le nom x() n'apparaisse dans le sous-programme.

Un même sous-programme peut ainsi être utilisé avec plusieurs tableaux. L'adresse du descripteur est ensuite transmise, comme dans l'exemple, et avec SWAP au début et à la fin, ces tableaux peuvent être appelés sous un seul nom.

Dans la version 3.0, il est conseillé de recourir plutôt à la transmission de tableau indirecte sous forme de paramètres VAR :

```
DIM x(3)
CHANGE (x())
PRINT x(2)
'
PROCEDURE change (VAR a())
  ARRAYFILL a(),1
RETURN
```

QSORT a(s) [,n] [,j%()]
QSORT x$(s) **WITH** i() [,n [,j%()]]

SSORT a(s) [,n] [,j%()]
SSORT x$(s) **WITH** i() [,n[,j%()]]

```
a()  : tableau quelconque, y compris de chaines
i()  : tableau entier (|,& ou %)
j%() : tableau entier de quatre octets
x$() : tableau de chaines
n    : iexp
s    : +,- ou aucun caractère
```

Les instructions SSORT et QSORT permettent de trier les éléments d'un tableau d'après leur taille. SSORT utilise à cet effet la méthode de tri *Shell sort*, QSORT la méthode *Quick sort*.

Un signe plus ou moins peut être placé entre les parenthèses suivant le nom du tableau à trier. Le signe moins indique que les éléments du tableau doivent être triés en ordre décroissant. Dans ce cas, après le tri, l'élément le plus grand figurera dans l'élément zéro du tableau. Le signe plus a pour effet de faire trier le tableau en ordre croissant. C'est donc la valeur la plus faible qui figurera après le tri dans l'élément zéro du tableau. Si aucun signe n'est spécifié, c'est comme si le signe plus avait été employé.

Le paramètre 'n' indique que seuls les 'n' premiers éléments du tableau doivent être triés (si OPTION BASE 0 s'applique, il s'agit des éléments d'indices 0 à n-1, sinon 1 à n).

Comme troisième paramètre peut être indiqué un autre tableau entier qui sera trié parallèlement au tri du premier tableau entier. Tout échange d'éléments dans le premier tableau entraînera un échange des mêmes éléments du second tableau. Cette possibilité peut être utilisée par exemple lorsque le premier tableau contient une clé de tri (par exemple le code postal) alors que plusieurs autres tableaux contiennent d'autres informations correspondantes. Le second tableau entier trié en même temps que le premier pourra par exemple contenir les indices du tableau, les indices d'un fichier "R", des valeurs LOC pour SEEK, etc...

Pour le tri de tableaux de chaînes de caractères, WITH permet d'indiquer un critère de tri sous forme d'un tableau comportant au moins 256 éléments. A défaut de WITH, c'est la table ASCII normale qui sera utilisée comme critère de tri (voyez le second exemple).

Exemples :

```
DIM x%(20)
PRINT "Non trié :                    ";
FOR i%=0 TO 10
  x%(i%)=RAND(9)+1
  PRINT x%(i%);"  ";
NEXT i%
PRINT
'
QSORT x%(),11
DIM index%(20)
PRINT "Tri décroissant :             ";
FOR i%=0 TO 10
  PRINT x%(i%);"  ";
  index%(i%)=i%
NEXT i%
PRINT
'
SSORT x%(-),11,index%()
PRINT "Tri croissant :               ";
FOR i%=0 TO 10
  PRINT x%(i%);"  ";
NEXT i%
PRINT
PRINT "Tableau trié en même temps :";
FOR i%=0 TO 10
  PRINT index%(i%);"  ";
NEXT i%
```

--> sort un tableau non trié et deux séries triées de nombres aléatoires. Dans une quatrième série figurent les valeurs d'un tableau trié parallèlement.

```
DIM b|(256)
FOR i%=0 TO 255
  b|(i%)=ASC(UPPER$(CHR$(i%)))
NEXT i%
FOR i%=1 TO 12
  READ a$,b$
  b|(ASC(a$))=ASC(b$)
NEXT i%
DATA à,A,â,A,é,E,è,E,ê,E,ë,E,î,I,ï,I,ô,O,ù,U,û,U,ç,C
DIM n$(3)
FOR i%=0 TO 3
  READ n$(i%)
NEXT i%
DATA Mercier,Mélasse,ça,ce
QSORT n$(),4
  PRINT "sans WITH b|():    ";
FOR i%=0 TO 3
  PRINT n$(i%),
NEXT i%
PRINT
QSORT n$() WITH b|(),4
  PRINT "avec WITH b|():    ";
FOR i%=0 TO 3
  PRINT n$(i%)
NEXT i%
```

--> Trie deux fois le tableau de chaînes de caractères n$(), une fois sans WITH, une fois avec. La spécification de WITH b|() a pour effet de faire classer les accents français comme les lettres normales correspondantes et les minuscules comme les majuscules.

INSERT x(i)=y
DELETE x(i)

```
(x : Nom d'un tableau)
(i : iexp)
(y : aexp ou sexp, suivant le type de variable du tableau)
```

Les instructions INSERT et DELETE permettent d'insérer ou de supprimer un élément d'un tableau. INSERT insère la valeur de l'expression numérique y en position i dans le tableau x. Tous les éléments du tableau dont l'indice est supérieur à i sont décalés d'une position vers le "haut". Si un élément figurait par exemple auparavant à la position i+3, il se retrouvera en position i+4 après l'instruction INSERT.

Le dernier élément du tableau est supprimé par cette opération d'insertion.

DELETE élimine l'élément i du tableau x. Tous les éléments du tableau dont l'indice est supérieur à i sont alors décalés d'une position vers le "bas". Le dernier élément du tableau est fixé sur 0 (ou sur une chaîne vide pour les tableaux de chaînes de caractères). Ces deux instructions conviennent particulièrement bien à la gestion de listes dans lesquelles on a en permanence à insérer ou à supprimer certains éléments.

Exemples :

```
DIM x%(5)
FOR i%=1 TO 5
  x%(i%)=i%
NEXT i%
INSERT x%(3)=33
FOR i%=0 TO 5
  PRINT x%(i%)
NEXT i%
```

--> affiche sur l'écran les nombres 0, 1, 2, 33, 3 et 4.

```
DIM x%(5)
FOR i%=1 TO 5
  x%(i%)=i%
NEXT i%
DELETE x%(3)
FOR i%=0 TO 5
  PRINT x%(i%)
NEXT i%
```

--> affiche sur l'écran les nombres 0, 1, 2, 4, 5 et 0.

VARIABLES RESERVEES

FALSE, TRUE, PI
DATE$, TIME$, SETTIME
TIMER

FALSE
TRUE
PI

Les deux constantes FALSE et TRUE contiennent les valeurs correspondant respectivement à logiquement faux (0) et logiquement vrai (-1).

La constante PI contient la valeur de la constante du cercle.

Exemple :

```
PRINT FALSE
IF TRUE
  PRINT PI
ENDIF
```

--> écrit les nombres 0 et 3.14159265359 sur l'écran.

DATE$
TIME$
SETTIME heure$,ladate$
**DATE$**=ladate$
**TIME$**=heure$

```
heure$,ladate$: sexp
```

La fonction DATE$ détermine la date système en format :

```
JJ.MM.AAAA (Jour.Mois.Année) ou MM/JJ/AAAA (Format US, voyez MODE).
```

TIME$ détermine l'heure système sous le format suivant :

```
HH:MM:SS  (Heures:Minutes:Secondes)
```

TIME$ se modifie à intervalles de 2 secondes.

L'instruction SETTIME permet de fixer la date et l'heure. Les chaînes *heure$* et la *date$* doivent alors présenter le format qui vient d'être décrit pour TIME$ et DATE$. Pour l'indication des années comprises entre 1980 et 2079, il n'est toutefois pas nécessaire de préciser le siècle. Si SETTIME est utilisée avec un format incorrect des chaînes *heure$* ou *ladate$*, les valeurs actuelles ne sont pas modifiées. La date et l'heure peuvent également être modifiées séparément avec DATE$= et TIME$=.

Exemple :

```
PRINT DATE$,TIME$
SETTIME "20:15:30","27.2.1988"
PRINT DATE$,TIME$
```

--> écrit sur l'écran les date et heure système actuelles, redéfinit la valeur de ces variables et les fait alors à nouveau afficher.

TIMER

La fonction TIMER fournit le temps écoulé depuis la mise en marche du système, en deux-centièmes de secondes.

Exemple :

```
t%=TIMER
FOR i%=1 TO 2500
NEXT i%
PRINT (TIMER-t%)/200
```

--> Indique en secondes le temps nécessaire à l'exécution de la boucle.

PARTICULARITES

LET, VOID, ~

LET permet d'affecter des valeurs à des variables dont les noms sont identiques à des mots d'instructions. VOID permet d'utiliser une fonction sans que la valeur qu'elle renvoie soit réutilisée.

LET x=y

```
(x : avar ou svar)
(y : aexp ou sexp)
```

LET permet d'affecter à une variable la valeur d'une expression. L'expression et la variable doivent être toutes deux numériques ou bien toutes deux du type chaîne de caractères. L'emploi de l'instruction LET est en principe superflu. Dans les anciens dialectes BASIC, elle servait à permettre d'utiliser des mots d'instructions comme variables. Mais GFA-BASIC reconnaît en principe automatiquement si un mot d'instruction est utilisé comme variable.

Exemple :

```
LET print=3
PRINT print
```

--> affiche le nombre 3 sur l'écran.

VOID fx
~fi

```
fx : aexp
fi : iexp
```

On distingue normalement, dans les langages de programmation, les instructions et les fonctions. Les instructions entraînent l'exécution d'une action quelconque. Les fonctions exécutent également une action, mais pour calculer une valeur qui est renvoyée à l'action en cours. Cette valeur en réponse peut ainsi elle-même n'être qu'un élément d'une expression, elle peut être affichée avec PRINT ou bien être affectée avec un signe égale à une variable, etc...

Dans de nombreux cas cependant, ce n'est pas la valeur en réponse qui intéresse le programmeur mais simplement l'action exécutée par la fonction. Prenons l'exemple de la fonction INP(2), qui renvoie le code de la touche actionnée sur le clavier. Lorsque le programme n'a pas besoin de savoir quelle touche a été actionnée parce qu'il attend simplement qu'une touche quelconque soit frappée, le code de la touche qui met fin à cette attente est sans importance. Dans ce type de situations, il est donc possible d'indiquer à GFA-BASIC, avec VOID, que la fonction doit être exécutée mais que la valeur en réponse peut être oubliée. Contrairement à ce qui se passe avec VOID, lorsque le *tilde* '~' est employé, c'est une expression entière qui est calculée puis oubliée. Avec INP(2) par exemple, VOID effectue en effet une conversion (superflue) en un nombre à virgule flottante. Le compilateur détecte automatiquement ces calculs superflus.

Exemple :

```
VOID INP(2)
```

ou

```
~INP(2)
```

--> Attend simplement qu'une touche soit actionnée sans l'affectation superflue fictive=INP(2). Le *tilde* permet à l'interpréteur de faire en plus l'économie de la conversion du résultat de INP(2) en une valeur à virgule flottante.

GESTION DE LA MEMOIRE

FRE
BMOVE
BASEPAGE, HIMEM
RESERVE
INLINE
MALLOC, MSHRINK, MFREE

FRE()
FRE(x)

```
x : aexp
```

Cette fonction calcule la place mémoire libre. Le paramètre x n'est pas utilisé. En outre, FRE(x) déclenche une *Garbage Collection* (regroupement dans le haut de la mémoire des zones de chaînes effectivement utilisées). FRE() fournit la taille de la zone de mémoire encore libre sans cette *Garbage Collection*.

Exemple :

```
libre%=FRE(0)
maxi%=libre%/3/4
DIM x%(maxi%)
PRINT libre%,maxi%
```

--> dimensionne un tableau de telle façon qu'il occupe un tiers de la place mémoire disponible. Un tableau entier de quatre octets occupe 4 octets par élément, d'où la division par 4.

BMOVE de_adr,en_adr,nombre

```
(de_adr,en_adr,nombre : iexp)
```

L'instruction BMOVE sert à copier des zones de mémoire. L'expression *de_adr* indique à quelle adresse commence la zone à copier ; *en_adr* indique à quelle adresse devra être placée la copie de la zone de mémoire ; *nombre* indique le nombre d'octets à copier.

Cette instruction est exécutée nettement plus rapidement pour les paramètres pairs que pour les paramètres impairs. Elle peut être utilisée même lorsque les zones d'origine et de copie se chevauchent.

Exemple :

```
DIM screen2%(64000/4)
adr%=VARPTR(screen2%(0))
FOR i%=0 TO 300 STEP 100
  PBOX 0,i%,639,i%+50
NEXT i%
BMOVE XBIOS(2),adr%,32000
BMOVE XBIOS(2),adr%+32000,32000
HIDEM
REPEAT
  IF MOUSEY<>my%
    BMOVE adr%+my%*80,XBIOS(2),32000
    my%=MOUSEY
  ENDIF
UNTIL MOUSEK=2
```

--> Les déplacements de la souris le long de l'axe des y entraînent un déplacement d'une zone de mémoire sur tout l'écran.

BASEPAGE
HIMEM

Dans la variable BASEPAGE figure l'adresse de la Basepage de l'interpréteur GFA-BASIC. La BASEPAGE est une zone de mémoire de 256 octets de long dont la structure est la suivante :

Octet	Contenu
0 à 3	Adresse du début de la TPA (transient program area)
4 à 7	Adresse de la fin de la TPA plus 1
8 à 11	Adresse du segment TEXT du programme
12 à 15	Longueur du segment TEXT
16 à 19	Adresse du segment DATA
20 à 23	Longueur du segment DATA
24 à 27	Adresse du segment BSS (block storage segment)
28 à 31	Longueur du segment BSS
32 à 35	Adresse de la DTA (disk transfer address)
36 à 39	Adresse de la Basepage du programme d'appel

40 à 43	Réservé
44 à 47	Adresse de la chaîne d'environnement
48 à 127	réservé
128 à 255	Ligne d'instruction (le premier octet indique la longueur du texte de l'instruction)

La variable HIMEM fixe à partir de quelle adresse la mémoire n'est plus employée par le GFA-BASIC. La valeur prédéfinie se situe 16384 octets en dessous de la mémoire écran.

Exemple :

```
a%=(BASEPAGE+&H2C)
DO
  a$=CHAR(a%)
  EXIT IF LEN(a$)=0
  PRINT a$
  ADD a%,SUCC(LEN(a$))     ! SUCC= +1
LOOP
```

--> Le chemin complet est affiché à l'écran.

RESERVE [n]

```
n : iexp
```

L'instruction RESERVE permet de fixer la taille que doit avoir la zone de mémoire employée par GFA-BASIC. Si n est positif, n octets seront réservés pour l'interpréteur, le reste étant libéré. Si n est négatif, l'effet sera le même qu'avec les deux instructions :

```
RESERVE
RESERVE FRE(0)-n
```

Si aucun paramètre n'est spécifié, la situation qui prévalait lors du lancement de l'interpréteur sera rétablie.

La zone de mémoire ne peut être réservée que par unités de 256 octets. Cette instruction peut être utilisée par exemple pour libérer une zone de mémoire pour les données ou pour les fichiers *Resource*.

Lorsque la zone de mémoire mise à la disposition de GFA-BASIC a été restreinte avec RESERVE, il ne faut pas oublier de l'augmenter à nouveau par la suite car la place mémoire disponible se restreindra sinon lors de chaque appel du programme.

Exemple :

```
RESERVE 2560
EXEC 0,"\WORDPLUS.PRG","",""
RESERVE
```

--> réserve 2560 octets, charge WORDPLUS.PRG (s'il est présent) et lance ce programme. Une fois WORDPLUS.PRG terminé, la place mémoire réservée est restituée au programme d'appel.

INLINE adr,nmb

```
adr : variable entière sur 4 octets mais pas un tableau
nmb : constante entière inférieure à 32700
```

Cette instruction ne peut être suivie d'aucun commentaire car la place mémoire voulue sera réservée sur un plan interne précisément à l'endroit où figure habituellement le commentaire. Cette place mémoire est annulée au départ (c'est-à-dire remplie d'octets nuls). Cette zone de mémoire commence toujours à une adresse paire.

Lorsque INLINE est exécutée, cette adresse est écrite dans la variable entière adr. Lors de la sauvegarde ou du chargement du programme, la zone de mémoire réservée est sauvegardée ou chargée en même temps.

Si on positionne le curseur sur la ligne de programme contenant l'instruction INLINE et qu'on actionne la touche *Help,* une ligne de menu apparaît alors dans la ligne la plus haute de l'éditeur, avec les entrées LOAD, SAVE et CLEAR. Les points "LOAD" et "SAVE" du menu permettent de charger des fichiers dans la ligne comportant l'instruction INLINE ou de sauvegarder des fichiers à partir de cette ligne. L'extension .INL est dans ce cas prédéfinie. Le point "CLEAR" du menu permet de supprimer la zone de mémoire réservée. Dans cette zone de mémoire peuvent être chargés des images, des tables ou des programmes assembleur.

Exemple :

Voyez l'exemple pour le C: dans la section consacrée aux routines système.

MALLOC(x)
MFREE(y)
MSHRINK(y,z)

```
x,y,z : iexp
```

La fonction MALLOC (GEMDOS 72) sert à réserver (allouer) des zones de mémoire. Si son paramètre x vaut -1, la fonction renvoie la longueur de la plus grande zone de mémoire d'un seul bloc encore disponible. MALLOC fait donc office dans ce cas de fonction de test.

Si x est un nombre positif, cela signifie que x octets doivent être réservés. Dans ce cas, MALLOC renvoie l'adresse de départ de la zone de mémoire réservée. Si une erreur est apparue lors de la tentative de réservation, 0 est renvoyé.

Lorsqu'il s'agit d'allouer des zones de mémoire assez importantes, il convient de restreindre avec RESERVE la place mémoire utilisée par GFA-BASIC. Il est vivement conseillé que les zones de mémoire allouées soient à nouveau libérées avant la fin du programme. Cette libération intervient d'ailleurs automatiquement lorsque l'interpréteur est abandonné.

MFREE (GEMDOS 73) libère à nouveau la place mémoire réservée avec MALLOC. y contient ici l'adresse de départ du bloc de mémoire à libérer, c'est-à-dire la valeur qui avait été renvoyée par MALLOC. La valeur renvoyée en réponse est 0 si la libération s'est effectuée sans erreur ou bien un numéro d'erreur négatif.

MSHRINK (GEMDOS 74) restreint une zone de mémoire qui avait été auparavant allouée avec MALLOC. Le paramètre y contient ici l'adresse de la zone de mémoire réservée, telle qu'elle avait été renvoyée en réponse par MALLOC. z contient la nouvelle longueur souhaitée pour le bloc de mémoire à raccourcir. La fonction MSHRINK renvoie 0 si le rétrécissement s'est effectué correctement, -40 si une adresse erronée a été spécifiée pour y et -67 si la nouvelle longueur souhaitée est supérieure à la longueur actuelle.

Il est important de veiller à ne jamais transmettre un pointeur erroné à MFREE et MSHRINK.

Exemple :

```
RESERVE 1000
PRINT MALLOC(-1)
adr%=MALLOC(60000)
PRINT adr%
IF adr%>0
  x%=MSHRINK(adr%,30000)
  y%=MFREE(adr%)
ENDIF
RESERVE
```

--> affiche sur l'écran la taille de la plus grande zone de mémoire d'un seul bloc. Affiche ensuite l'adresse d'une zone de mémoire réservée de 60000 octets, la ramène (si la réservation s'est effectuée sans problème) à 30000 puis la libère totalement.

3. LES OPERATEURS

Les opérateurs sont les éléments d'un langage qui servent à combiner et à comparer des expressions numériques, logiques ou des expressions de chaînes de caractères. Ce chapitre vous présentera les opérateurs de GFA-BASIC 3.0 dans l'ordre suivant :

La première section traitera des opérateurs numériques (+, -, *, /, ^, DIV, \, MOD). Ces opérateurs combinent deux expressions numériques pour produire un nombre qui peut par exemple être affecté à une variable à l'aide du signe égal. Les opérateurs + et - ont encore une seconde fonction car ils peuvent aussi être utilisés comme signe.

La seconde section est consacrée aux opérateurs logiques (AND, OR, XOR, NOT, IMP, EQV). Ils combinent deux expressions logiques pour renvoyer une valeur de vérité (vrai ou faux). L'opérateur NOT est particulier à cet égard car il n'agit que sur une seule expression dont il inverse la valeur de vérité (vrai devient faux et inversement).

La section 3 décrit l'opérateur de chaînes de caractères. Il s'agit du signe plus qui permet de fusionner deux expressions de chaînes de caractères.

La section suivante décrit les opérateurs de comparaison. Ces opérateurs permettent de comparer deux expressions numériques ou deux expressions de chaînes de caractères. Le résultat obtenu est une valeur booléenne (vrai ou faux).

La dernière section décrit enfin dans quel ordre les opérateurs sont traités lorsqu'une expression numérique ou logique en comprend plusieurs. C'est ce qu'on appelle la hiérarchie des opérateurs. Nous y expliquerons également le rôle des parenthèses "(" et ")" qui permettent de modifier l'ordre de traitement fixé par cette hiérarchie.

OPERATEURS ARITHMETIQUES

```
+ - * / ^ DIV \ MOD
+ -
```

Les opérateurs arithmétiques +, -, *, /, ^, DIV, \ et MOD combinent deux expressions numériques pour produire un nombre.

Ce nombre peut lui-même être un élément d'une expression numérique ou logique, être affecté à une variable ou encore être affiché, par exemple avec PRINT. Les opérateurs + et - sont également utilisés comme signes.

x+y	:	produit la somme des nombres x et y (Addition).
x-y	:	produit la différence de x moins y (Soustraction).
x*y	:	produit le produit de x par y (Multiplication).
x/y	:	produit le quotient de x divisé par y (Division).
x^y	:	produit la puissance y d'un nombre x (Elévation à la puissance).
x DIV y x \ y	:	fournit la partie entière du résultat de la division de x divisé par y (Division entière).
x MOD y	:	fournit le reste de la division de x par y (Calcul Modulo).

Les équivalences suivantes s'appliquent à DIV et MOD :

```
x DIV y = TRUNC(x/y)
x MOD y = x-y*TRUNC(x/y)
```

Exemples :

```
13 DIV 4   produit 3
13 MOD 4   produit 1
```

+x : dote la valeur x d'un signe positif, ce qui donne donc x.
-x : dote la valeur x d'un signe négatif. Le nombre produit aura donc un signe négatif si x était positif et un signe positif si x était négatif.

OPERATEURS LOGIQUES

AND OR XOR NOT IMP EQV

Ces opérateurs logiques travaillent au niveau des bits pour les valeurs entières sur 32 bits.

Les opérateurs logiques combinent deux expressions logiques pour produire une valeur booléenne (vrai ou faux).

L'opérateur NOT constitue cependant une exception à cet égard car il inverse la valeur de vérité de l'expression qui le suit.

La valeur numérique correspondant à logiquement faux (FALSE) est 0. Toute autre valeur numérique est interprétée comme logiquement vraie. La variable TRUE vaut -1. Tous les opérateurs logiques peuvent aussi être appliqués à des expressions numériques. Dans ce cas, les opérations logiques sont exécutées bit par bit.

Les tables dites de vérité représentent certainement le meilleur moyen de décrire l'effet des opérateurs logiques. Ces tables présentent différentes colonnes dont les premières contiennent les valeurs de vérité des expressions à combiner (les opérandes), la dernière colonne contenant la valeur logique produite.

NOT x

```
x : iexp
```

L'opérateur NOT (Négation) inverse l'expression logique qui le suit. C'est le seul opérateur qui n'attende qu'un seul argument.

Il modifie (donc inverse) chaque bit de son argument.

x	NOT x
v	f
f	v

Exemples :

```
PRINT NOT FALSE
PRINT NOT TRUE
PRINT NOT 0
```

--> Les nombres -1, 0, et -1 apparaissent sur l'écran.

```
x=1
PRINT BIN$(x,2)
PRINT BIN$(NOT x,2)
```

--> 01 et 10 apparaissent sur l'écran.

```
x%=17
PRINT BIN$(x%,8),x%
PRINT BIN$(NOT x%,8),NOT x%
```

--> Sortie :
00010001 17
11101110 -18

x AND y

```
x,y : iexp
```

L'opérateur logique AND (Conjonction) examine si deux expressions x et y sont vraies toutes deux. Dans ce cas seulement la valeur produite sera TRUE (logiquement vrai, -1). Si l'une des deux expressions logiques est fausse ou si les deux expressions logiques sont fausses, AND produira un logiquement faux.

AND combine chacun des 32 bits de ses arguments.

x	y	x AND y
v	v	v
v	f	f
f	v	f
f	f	f

Exemples :

```
PRINT TRUE AND -1
PRINT FALSE AND TRUE
```

--> Sur l'écran apparaissent les nombres -1 et 0.

```
x=3
y=10
PRINT BIN$(x,4)
PRINT BIN$(y,4)
PRINT BIN$(x AND y,4),x AND y
```

--> 0011, 1010, 0010 et le nombre 2 apparaissent sur le moniteur.

x **OR** y

```
x,y : iexp
```

OR (Disjonction) examine si l'une au moins des deux expressions logiques x et y est vraie. Dans ce cas, un logiquement vrai est produit. Ce n'est que si x et y sont toutes deux logiquement fausses que x OR y produira également un logiquement faux. Contrairement à ce qui se passe avec XOR, un logiquement vrai est également produit dans le cas où x et y sont toutes deux vraies.

OR produit vrai lorsque l'un au moins de ses deux arguments est vrai. OR travaille également au niveau des bits.

x	y	x OR y
v	v	v
v	f	v
f	v	v
f	f	f

Exemples :

```
PRINT TRUE OR -1
PRINT FALSE OR TRUE
PRINT 0 OR FALSE
```

--> Les nombres -1, -1 et 0 apparaissent sur l'écran.

```
x=3
y=10
PRINT BIN$(x,4)
PRINT BIN$(y,4)
PRINT BIN$(x OR y,4),x OR y
```

--> 0011
1010
1011 et le nombre 11 apparaissent sur l'écran.

x **XOR** y

```
x,y : iexp
```

Cette opérateur examine si une et une seule des deux expressions logiques x et y est logiquement vraie. Dans ce seul cas, x XOR y renverra également logiquement vrai comme résultat. Si x et y sont toutes deux logiquement vraies ou bien toutes deux logiquement fausses, XOR renvoie un logiquement faux.

La différence avec OR est donc que TRUE OR TRUE est vraie alors que TRUE XOR TRUE est fausse (XOR = disjonction exclusive).

XOR produira vrai si (et seulement si) un seul de ses arguments est vrai. XOR traite également séparément chacun des 32 bits de ses arguments.

x	y	x XOR y
v	v	f
v	f	v
f	v	v
f	f	f

Exemples :

```
PRINT FALSE XOR -1
PRINT -1 XOR 1
PRINT 0 XOR FALSE
```

--> Les nombres -1, 0 et 0 apparaissent sur le moniteur.

```
x=3
y=10
PRINT BIN$(x,4)
PRINT BIN$(y,4)
PRINT BIN$(x XOR y,4),x XOR y
```

--> 0011
1010
1001 et le nombre 9 apparaissent sur l'écran.

x IMP y

```
x,y : iexp
```

L'opérateur IMP (Implication) équivaut à une déduction logique. Une déduction logique ne peut avoir été fausse que si une affirmation vraie est suivie de quelque chose de faux. x IMP y ne sera donc fausse que si x est vraie et y fausse.

IMP travaille au niveau des bits. Contrairement à AND, OR, XOR et EQV, l'ordre des arguments joue ici un rôle déterminant.

x	y	x IMP y
v	v	v
v	f	f
f	v	v
f	f	v

Exemples :

```
PRINT TRUE IMP -1
PRINT 0 IMP FALSE
PRINT TRUE IMP 0
```

--> Les nombres -1, -1 et 0 apparaissent sur l'écran.

```
x=3
y=10
PRINT BIN$(x,4)
PRINT BIN$(y,4)
PRINT BIN$(x IMP y,4)
```

--> 0011
1010
1110 apparaissent sur l'écran.

x **EQV** y

```
x,y : iexp
```

L'opérateur EQV (Equivalence) produit logiquement vrai lorsque les expressions x et y ont la même valeur de vérité.

EQV travaille au niveau des bits et fixe les bits qui sont identiques dans les deux arguments. C'est exactement le contraire de ce que fait XOR, de sorte que (x EQV y) équivaut à (NOT x XOR y).

A	B	A EQV B
v	v	v
v	f	f
f	v	f
f	f	v

Exemple :

```
PRINT TRUE EQV FALSE
PRINT FALSE EQV FALSE
```

--> Sur l'écran apparaissent les nombres 0 et -1.

```
x=3
y=10
PRINT BIN$(x,4)
PRINT BIN$(y,4)
PRINT BIN$(x EQV y,4)
```

--> 0011
1010
0110 apparaissent sur l'écran.

OPERATEUR DE CHAINES DE CARACTERES

a$+b$

```
(a$,b$ : sexp)
```

L'opérateur + permet également de combiner des chaînes de caractères, le résultat étant une chaîne de caractères composée de a$ et b$.

Exemple :

```
a$="GFA-"
PRINT a$+"BASIC"
```

--> Sur l'écran apparaît le texte 'GFA-BASIC'.

OPERATEURS DE COMPARAISON

```
= == >= <= <>
```

Les opérateurs de comparaison permettent de comparer des expressions numériques, logiques ou des expressions de chaînes de caractères. Le résultat de cette comparaison est toujours une valeur logique (vrai = -1 ou faux = 0). L'opérateur == constitue une exception à cet égard car il ne permet pas de comparer des expressions de chaînes de caractères.

x=y

```
x,y : exp
```

L'opérateur = examine si deux expressions numériques ou deux expressions de chaînes de caractères sont identiques. Si les deux expressions sont identiques, un logiquement vrai est produit, sinon un logiquement faux.

Exemples :

```
x=6
IF 2=x/3
  PRINT "Ok"
ENDIF
PRINT 2=x/3
```

--> Ok et -1 apparaissent sur l'écran.

```
a=SINQ(77)
b=SIN(RAD(77))
PRINT a=b
PRINT a==b
```

--> Les nombres 0 (pour logiquement faux) et -1 (pour logiquement vrai) apparaissent.

x= =y

```
(x,y : aexp)
```

Examine si deux expressions numériques sont à peu près égales. Cette comparaison ne prend en compte que 8,5 chiffres après la virgule (28 bits de la mantisse du nombre à virgule flottante). Il est intéressant d'utiliser = = lorsqu'on emploie des nombres à virgule flottante qui peuvent entraîner des imprécisions d'arrondissement.

Exemples :

```
PRINT 1.0000000001=1
PRINT 1.0000000001==1
```

--> Les nombres 0 et -1 apparaissent sur l'écran.

```
PRINT 2^2^0.5
IF 2=2^2^0.5
  PRINT "Vrai"
ELSE
  PRINT "Faux"
ENDIF
IF 2==2^2^0.5
```

```
    PRINT "Vrai"
  ELSE
    PRINT "Faux"
  ENDIF
```

--> Le nombre 2.000000000033 et les textes 'Faux' et 'Vrai' apparaissent sur l'écran.

```
x<y
x>y
x< =y
x> =y
```

```
(x,y : exp)
```

Ces opérateurs servent à comparer des expressions numériques et des expressions de chaînes. Pour les expressions numériques, la comparaison s'effectue entre les nombres calculés à partir de ces expressions.

Pour les expressions de chaînes, la comparaison s'effectue d'après le code ASCII des caractères. La chaîne "ABC" est traitée comme une séquence des nombres 65,66 et 67. Si l'affirmation testée est "ABC">"AAA", ce seront tout d'abord les premiers caractères des deux chaînes qui seront comparés, qui ont tous deux le code ASCII 65. Le caractère suivant de chaque chaîne sera alors à son tour pris en compte. Comme B (code ASCII 66) porte un code supérieur à A, B sera considéré comme "supérieur". La comparaison des deux chaînes pourra donc s'arrêter ici et le résultat de cette affirmation sera logiquement vrai puisque "ABC" est bien supérieur à "AAA".

Un cas particulier de comparaison de chaînes de caractères se présente lorsqu'une des deux chaînes se termine avant que des caractères différents n'aient été identifiés. Par exemple avec l'affirmation "AA">"A". Cette affirmation est logiquement vraie car un caractère qui n'existe pas est considéré par définition comme le "plus petit caractère" possible. Même "A"+Chr$(0)>"A" sera logiquement vrai.

Venons-en maintenant aux différents opérateurs :

```
 x>y   est vrai si x est supérieur à y
 x<y   est vrai si x est inférieur à y
x>=y   est vrai si x est supérieur ou égal à y
x<=y   est vrai si x est inférieur ou égal à y
```

Les écritures x<=y et x=<y ainsi que x>=y et x=>y sont équivalentes. x><y est converti en x<>y.

Exemple :

```
PRINT "AAA">"aaa"
PRINT -1<=4-5
```

--> Les nombres 0 et -1 apparaissent sur l'écran.

x< >y

```
(x,y : exp)
```

Cet opérateur teste si deux expressions numériques ou deux expressions de chaînes sont différentes. Si c'est le cas, alors l'affirmation x<>y sera logiquement vraie. Si x et y sont identiques, alors x<>y sera logiquement fausse. Les écritures x<>y et x><y sont équivalentes.

Exemple :

```
PRINT "Test"<>"test"
PRINT -1<>4-5
```

--> Les nombres -1 et 0 apparaissent sur l'écran.

OPERATEUR D'AFFECTATION

x=y

```
x : var
y : exp
```

Le signe égal peut être utilisé non seulement comme opérateur de comparaison, comme nous l'avons vu dans les sections précédentes, mais aussi comme opérateur d'affectation. Dans ce cas, la valeur de l'expression y placée à droite du signe égal est calculée pour être affectée à la variable x placée à gauche de ce signe.

Les expressions numériques ne peuvent être affectées qu'à des variables numériques, les expressions de chaînes ne peuvent de même être affectées qu'à des variables de chaînes de caractères.

Il existe également une instruction d'affectation équivalente, LET var = exp (voyez LET), qui permet l'affectation à des variables comme *let* ou *rem* par exemple.

Exemple :

```
x=LEN("TEST")+3
a$="GF"+CHR$(65)
PRINT x,a$
```

--> Sur l'écran apparaissent 7 et 'GFA'.

HIÉRARCHIE DES OPÉRATEURS

Lorsqu'une expression comprend différents opérateurs, ceux-ci sont traités dans un ordre bien défini. Cet ordre dépend de leur rang dans ce qu'on appelle la hiérarchie des opérateurs. Les opérateurs placés tout en haut de cette hiérarchie sont naturellement les premiers traités.

Cette hiérarchie est la suivante :

( )	Parenthèses
+	Addition de chaînes de caractères
= < > => <= <>	Comparaison de chaînes de caractères
+ -	Signe
^	Elévation à la puissance
* /	Multiplication, Division
DIV MOD	Division entière et Modulo
+ -	Addition, soustraction
= == < <= >= > <>	Opérateurs de comparaison
AND OR XOR IMP EQV	Opérateurs logiques
NOT	Négation

L'emploi des parenthèses permet de faire traiter des opérateurs avant d'autres dont le rang hiérarchique est pourtant plus élevé.

Exemple :

```
PRINT 2+4*3
PRINT (2+4)*3
PRINT 2+(4*3)
PRINT 3*2^2
```

--> Sur l'écran apparaissent les nombres 14, 18, 14 et 12.

4. FONCTIONS NUMERIQUES

FONCTIONS MATHEMATIQUES

(ABS, SGN)
(ODD, EVEN)
(INT, TRUNC, FIX, FRAC, ROUND)
(MAX, MIN)
(SQR)
(EXP, LOG, LOG(10))
(SIN, COS, TAN, ASIN, ACOS, ATN, DEG, RAD)
(SINQ, COSQ)

Les fonctions numériques suivantes sont disponibles : les fonctions numériques ABS et SGN fournissent la valeur absolue et le signe d'une expression numérique. ODD et EVEN examinent si un nombre est pair ou impair. INT, TRUNC, FIX et FRAC fournissent les chiffres avant ou les chiffres après la virgule des expressions numériques. ROUND arrondit une expression.

MAX et MIN fournissent respectivement la plus grande ou la plus petite expression numérique parmi une liste de ces expressions. SQR renvoie la racine carrée d'une expression. Les fonctions trigonométriques sont ASIN, ACOS, SIN, COS, TAN et ATN. EXP et LOG calculent les logarithmes et les puissances du nombre d'Euler. LOG10 calcule le logarithme de base 10.

DEG et RAD servent à la conversion des degrés en radians. RND, RANDOM, RAND et RANDOMIZE servent à produire des nombres aléatoires.

ABS(x)
SGN(x)

```
(x : aexp)
```

La fonction ABS fournit la valeur absolue d'une expression numérique. Elle renverra donc les valeurs de réponse suivantes :

Si x est :		ABS(x) sera :
négatif	-->	-x
égal à 0	-->	0
positif	-->	x

La fonction SGN permet de déterminer le signe d'une expression numérique.

Si x est :		SGN(x) sera :
négatif	-->	-1
égal à 0	-->	0
positif	-->	1

Exemple :

```
x=-2
y=ABS(x)
PRINT SGN(x),ABS(5-3),SGN(ABS(x*3))
```

--> Sur l'écran apparaissent les nombres -1, 2 et 1.

ODD(x)
EVEN(x)

```
x : aexp
```

Ces deux fonctions examinent si l'expression numérique x est paire ou impaire. ODD renvoie la valeur -1 (TRUE) si x est impair et 0 (FALSE) si x est pair. EVEN renvoie -1 pour un x pair et 0 pour un x impair. 0 est traité comme un nombre pair.

Si x est :		ODD(x) sera	EVEN(x) sera :
pair	-->	0 (FALSE)	-1 (TRUE)
impair	-->	-1 (TRUE)	0 (FALSE)
égal à 0	-->	0 (FALSE)	-1 (TRUE)

Exemple :

```
x=2
PRINT ODD(x),EVEN(-2),ODD(3*5), EVEN(-3+x)
```

--> Les nombres 0, -1, -1 et 0 apparaissent sur l'écran.

INT(x)
TRUNC(x)
FIX(x)
FRAC(x)

```
(x : aexp)
```

Ces fonctions extraient les chiffres avant ou après la virgule dans des expressions numériques. INT, TRUNC (et la fonction FIX, identique à TRUNC) fournissent un nombre entier. La fonction TRUNC tronque tout simplement les chiffres après la virgule de x. INT fournit le plus grand nombre entier inférieur ou égal à x. Il n'y a naturellement aucune différence pratique entre TRUNC (FIX) et INT pour les valeurs positives de x. La différence apparaît cependant dès que x est négatif et qu'il comprend des chiffres après la virgule. C'est ainsi qu'avec -1.2 pour x, TRUNC éliminerait le chiffre après la virgule et produirait -1 comme résultat. INT rechercherait par contre le nombre entier inférieur ou égal à -1.2 le plus proche et le résultat serait donc -2.

FRAC renvoie les chiffres après la virgule de x, en éliminant donc les chiffres avant la virgule. La valeur fournie par FRAC est de même signe que x. FRAC est complémentaire de TRUNC mais non de INT. En effet :

```
x = TRUNC(x) + FRAC(x)
```

Si TRUNC est remplacée par INT, l'équation devient fausse dans le domaine des nombres négatifs (par exemple pour y = -1.2).

Exemple :

```
x=-1.4
y=TRUNC(1.3)
PRINT y,INT(x),FIX(3*x),FRAC(x-3)
```

--> Sur l'écran apparaissent les nombres 1, -2, -4 et -0.4.

ROUND(x,[n])

```
x : aexp
n : iexp
```

La fonction ROUND(x) renvoie une valeur arrondie. La variante ROUND(x,n) arrondit l'expression 'x' à 'n' chiffres après la virgule. Si 'n' vaut zéro, la fonction arrondit aux nombres entiers, comme ROUND(x). Si 'n' est négatif, l'arrondissement se fait avant la virgule. ROUND(155,-1) donnera par exemple le nombre 160 (cf. aussi CINT()= arrondissement avec résultat entier).

Exemples :

```
y=ROUND(-1.2)
PRINT y,ROUND(1.7)
```

--> Affiche -1 et 2 sur l'écran.

```
FOR i%=-5 TO 5
  PRINT i%,ROUND(PI*100,i%)
NEXT i%
```

--> Affiche sur l'écran la variable de comptage et l'expression correspondante formatée. La constante Pi sera donc multipliée par 100 et arrondie à i% chiffres. Si i% est négatif, l'arrondissement se fera avant la virgule.

MIN(x [,y,z,...])
MIN(x$ [,y$,z$,...])
MAX(x [,y,z,...])
MAX(x$ [,y$,z$,...])

```
(x,y,z : aexp)
(x$,y$,z$ : sexp)
```

La fonction MAX renvoie la plus grande des expressions numériques x,y,z,...., énumérées dans la liste des paramètres de cette fonction. MIN permet de même d'obtenir l'expression la plus petite. Les fonctions MIN et MAX peuvent être appliquées de la même façon aux chaînes de caractères.

Exemple :

```
x=3
y=MAX(3,5,5-4)
PRINT MIN(x,y),MAX(-1,x*2)
```

--> Sur l'écran apparaissent les nombres 3 et 6.

SQR(x)

```
(x : aexp)
```

Renvoie la racine carrée de l'expression numérique x. Si l'expression x est inférieure à zéro, cela entraînera un message d'erreur.

Exemples :

```
x=9
y=SQR(x)
PRINT y,SQR(4*4)
```

--> Les nombres 3 et 4 apparaissent sur l'écran.

```
PRINT SQR(SQR(16))
PRINT SQR(-2)
```

--> Le nombre 2 apparaît sur l'écran, puis un message d'erreur indiquant qu'il est impossible d'obtenir la racine carrée d'un nombre négatif.

EXP(x)
LOG(x)
LOG10(x)

```
x : aexp
```

EXP calcule la puissance x dont la base est le nombre d'Euler (e=2.1782818284...) et LOG fournit la fonction inverse, c'est-à-dire calcule le logarithme de x de base e (logarithme naturel). LOG10 calcule de même le logarithme de x de base 10 (logarithme décimal).

L'expression numérique x doit être supérieure à zéro pour les fonctions logarithmiques, faute de quoi un message d'erreur apparaît.

Exemple :

```
x=2
y=EXP(2)
PRINT y,LOG10(2*5),LOG(x)
```

--> Sur l'écran apparaissent 7.389056098931, 1 et 0.6931471805599.

SIN(x)
COS(x)
TAN(x)

ASIN(x)
ACOS(x)
ATN(x)

DEG(),
RAD()
SINQ(degrés)
COSQ(degrés)

```
x,degrés : aexp
```

Ces fonctions sont les fonctions trigonométriques. L'expression numérique x est interprétée comme mesure en radians. Les fonctions calculent respectivement :

```
SIN  --> le sinus
COS  --> le cosinus
TAN  --> la tangente
ASIN --> le sinus d'arc
ACOS --> le cosinus d'arc
ATN  --> la tangente d'arc
```

Pour convertir un angle de radians en degrés, on utilise la fonction DEG() ou sa fonction inverse RAD(). DEG(x) équivaut ici à (x * 180 / PI).
Les fonctions SINQ et COSQ fournissent des valeurs de sinus ou cosinus interpolées, une table des valeurs sinus par intervalles de degrés (interne à GFA BASIC) étant utilisée à cet effet.

Suivant la valeur de l'argument degrés de la fonction, les valeurs intermédiaires sont interpolées de façon linéaire par pas de 1/16 degrés. Cette précision est suffisante pour l'affichage sur l'écran et elle aboutit à des valeurs ne se distinguant pas de celles calculées avec SIN ou COS, tout en étant beaucoup plus rapide (à peu près dix fois plus rapide).

Contrairement à SIN et COS, SINQ et COSQ attendent leurs paramètres en degrés.

`SINQ(degrés)`	équivaut à	`SIN(RAD(degrés))`
`COSQ(degrés)`	équivaut à	`COS(RAD(degrés))`

Exemples :

```
x=90
y=COS(x*PI/180)
z=270*PI/180
PRINT y,SIN(z),TAN(45),ATN(1/2)
```

--> Les nombres 1,-1,1.619775190544 et 0.4636476090008 apparaissent.

```
alpha%=30
PRINT SINQ(alpha%)
```

--> Affiche 0.5 sur l'écran.

GENERATEUR DE NOMBRES ALEATOIRES

Les fonctions

RND [(x)]
RANDOM(x)
RAND(y)

et l'instruction

RANDOMIZE y

```
x : aexp
y : iexp
```

Ce groupe d'instructions sert à produire des nombres aléatoires. RND produit un nombre aléatoire entre 0 et 1 (0 inclus, 1 exclu). Le paramètre facultatif x ne sert à rien.

RANDOM produit un nombre aléatoire entier entre 0 et x (0 inclus, x exclu). L'expression numérique x peut ne pas être entière si vous souhaitez que tous les nombres n'apparaissent pas avec la même probabilité.

RAND produit un nombre entier de 16 bits compris entre 0 et y-1. 16 bits de y sont évalués.

L'instruction RANDOMIZE initialise le générateur de nombres aléatoires sur la valeur 'y'. Si le générateur de nombres aléatoires est initialisé plusieurs fois sur la même valeur 'y', la même séquence de nombres aléatoires sera produite chaque fois. Au début de chaque exécution du programme, le générateur de nombres aléatoires est initialisé sur un nombre sélectionné "au hasard". Si vous n'utilisez pas RANDOMIZE, vous obtiendrez donc avec RND, RANDOM ou RAND des nombres aléatoires différents lors de chaque lancement du programme.

Pour initialiser le générateur de nombres aléatoires, vous pouvez utiliser RANDOMIZE sans paramètre ou bien RANDOMIZE 0.

Exemples :

```
x=RND
PRINT x,RANDOM(10)
```

--> Deux nombres aléatoires apparaissent sur le moniteur.

```
x=RANDOM(2)
y=RAND(4)
PRINT x,y,RAND(x),RANDOM(3*x)
```

--> Quatre nombres aléatoires entiers apparaissent sur l'écran.

```
RANDOMIZE 3
x=RND
RANDOMIZE 3
PRINT x,RND
```

--> Le même nombre "aléatoire" apparaît deux fois sur le moniteur.

ARITHMETIQUE ENTIERE

INSTRUCTIONS ET FONCTIONS

(DEC, INC)
(ADD, SUB, MUL, DIV)
(PRED(), SUCC())
(ADD(), SUB(), MUL(), DIV(), MOD())

(DEC, INC)
(ADD, SUB, MUL, DIV)

Ces instructions sont autant d'abréviations des expressions suivantes :

`DEC x`	équivaut à	`x=x-1`
`INC x`	équivaut à	`x=x+1`
`ADD x,y`	équivaut à	`x=x+y`
`SUB x,y`	équivaut à	`x=x-y`
`MUL x,y`	équivaut à	`x=x*y`
`DIV x,y`	équivaut à	`x=x/y`

Les instructions désignées à gauche sont exécutées nettement plus vite que les expressions apparaissant à droite. Cette différence est particulièrement importante pour les variables entières.

Il importe de noter que INC, DEC, ADD, SUB, MUL et DIV n'effectuent aucun contrôle de débordement pour les variables entières (%,&,|).

DEC i
INC i

```
i : avar
```

Les instructions DEC et INC servent à modifier une valeur d'une unité. DEC diminue (décrémente) la valeur de la variable numérique i d'une unité, INC augmentant (incrémentant) cette valeur d'une unité.

Ces instructions travaillent aussi avec les variables à virgule flottante mais sont beaucoup plus rapides avec les variables entières.

Exemples :

```
x%=4
y%=7
DEC x%
INC y%
PRINT x%,y%
```

--> Sur l'écran apparaissent les nombres 3 et 8.

```
a|=255
INC a|
INC a|
PRINT a|
```

--> Aura 1 pour résultat puisque 255 est la valeur maximale que peut avoir une variable entière sur 1 octet.

ADD x,y
SUB x,y
MUL x,y
DIV x,y

```
(x : avar)
(y : aexp)
```

L'instruction ADD augmente la variable x de la valeur y, alors que SUB x la diminue de la valeur y. MUL multiplie x par y et affecte le résultat à x. DIV divise la variable x par y et place le résultat dans x.

Pour ces instructions, x doit être une variable numérique, y étant une expression numérique. Ces instructions travaillent aussi avec les variables à virgule flottante mais sont beaucoup plus rapides avec les variables entières.

Exemple :

```
x%=1
y%=2
z%=3
ADD x%,y%          !x vaut maintenant 3
SUB z%,(x%-1)/2    !L'expression numérique (x%-1)/2 donnera 1
PRINT x%,y%
```

--> Sur l'écran apparaissent les nombres 3 et 2.

(PRED(), SUCC())
(ADD(), SUB(), MUL(), DIV(), MOD())

PRED et SUCC renvoient respectivement le prochain nombre supérieur ou le prochain nombre inférieur. ADD(), SUB(), MUL(), DIV() et MOD() permettent une arithmétique entière rapide en notation polonaise.

PRED(i)
SUCC(i)

```
i : iexp
```

PRED renvoie le prochain nombre inférieur, SUCC le prochain nombre supérieur. PRED fournit en quelque sorte le prédécesseur et SUCC le successeur d'une expression numérique. Ces deux fonctions travaillent en arithmétique entière, les chiffres après la virgule étant donc ignorés.

Les deux fonctions peuvent également être appliquées aux expressions de chaînes (voyez également la section consacrée à la gestion des chaînes de caractères).

Ces deux fonctions travaillent aussi avec les variables à virgule flottante mais sont beaucoup plus rapide avec les variables entières.

Exemple :

```
i%=6
j%=PRED(i%)
PRINT j%,SUCC(2),PRED(3*i%)
```

--> Sur l'écran apparaissent les nombres 5, 3 et 17.

ADD(x,y)
SUB(x,y)
MUL(x,y)
DIV(x,y)
MOD(x,y)

```
(x,y : iexp)
```

Ces fonctions peuvent être utilisées à la place des opérateurs numériques suivants :

`ADD(x,y)`	équivaut à	`x+y`
`SUB(x,y)`	équivaut à	`x-y`
`MUL(x,y)`	équivaut à	`x*y`
`DIV(x,y)`	équivaut à	`x\y` `x DIV y`
`MOD(x,y)`	équivaut à	`x MOD y`

Les fonctions indiquées ci-dessus travaillant avec l'arithmétique entière, les chiffres après la virgule sont ignorés. Après les instructions

```
x%=5
y%=4
ADD y%,3
z%=SUB(x%,3)
PRINT x%, y% ,z%
```

x% vaudra 5, y% 7 et z% 2.

Les fonctions ADD, SUB, MUL, DIV et MOD peuvent être imbriquées à volonté. Cet ordre des opérateurs et des opérandes est appelé *notation polonaise*.

Exemples :

```
DEFINT "a-z"
x=4
y=ADD(x,x)   !y vaudra 8
z=SUB(x,2)
PRINT y,z,ADD(x,MUL(y,2))
```

--> Sur l'écran apparaissent les nombres 8, 2 et 20.

```
DEFINT "a-z"
x=2
y=MUL(x,3)                    !y vaudra 6
PRINT y,DIV(8,x),MOD(11,4)  !MOD(11,4) vaut 3
```

--> Sur l'écran apparaissent les nombres 6, 4 et 3.

```
DEFINT "a-z"
x=5
y=ADD(SUB(x,2),MUL(3,4))
PRINT y,DIV(8,MOD(14,4))
```

--> Sur l'écran apparaissent les nombres 15 et 4.

OPERATIONS DE BITS

(BCLR, BSET, BTST, BCHG)
(SHL, SHR, ROL, ROR)
(AND(), OR(), XOR(), IMP(), EQV())
(SWAP())
(BYTE(), CARD(), WORD())

Les opérations de bits modifient des expressions numériques au niveau des bits. Les instructions (bien connues des programmeurs en assembleur) BCLR, BSET, BTST et BCHG servent à annuler, fixer, tester et inverser des bits, SHL, SHR, ROL et ROR servent à décaler des bits ou à effectuer une rotation de bits.

Les fonctions AND, OR, XOR, IMP et EQV sont des combinaisons logiques. La numérotation des bits obéit à la convention suivante : 0 est le bit de plus faible poids ; pour les valeurs entières de 4 octets, 31 est le bit de plus fort poids en même temps que le bit de signe (lorsque le bit de signe est mis, c'est un nombre négatif en complément de 2 qui sera représenté, sinon un nombre positif).

SWAP échange les deux mots d'une valeur sur quatre octets. BYTE lit les 8 bits inférieurs et CARD les 16 bits inférieurs d'une expression entière. WORD étend un mot à un long mot, c'est-à-dire que le bit 15 est copié dans les bits 16 à 31.

BCLR(x,y)
BSET(x,y)
BCHG(x,y)
BTST(x,y)

```
(x,y : iexp)
```

Ces instructions permettent d'annuler, de fixer, d'inverser et de tester un bit. La numérotation des bits commence par zéro. Le numéro de bit est compris entre 0 et 31 et est masqué avec AND 31 sur un plan interne au processeur.

La fonction BCLR annule (fixe sur zéro) le y-ième bit de l'expression numérique x. BSET fixe de même sur 1 le bit numéro y de x. BCHG fixe le bit y de x sur 1 s'il valait 0 auparavant ou sur 0 s'il valait auparavant 1. La fonction BTST donnera -1 (TRUE) si le bit y de x égale 1 et 0 (FALSE) si ce bit égale 0.

Exemples :

```
x=BSET(0,3)
PRINT x,BSET(0,5)
```

--> Affichage des nombres 8 et 32.

```
REPEAT
  t|=Inp(2)
  PRINT CHR$(t|),CHR$(BCLR(t|,5))
UNTIL CHR$(t|)="x"
```

--> Si vous appuyez sur une touche de lettre, la lettre minuscule ainsi que la majuscule correspondante apparaissent (le bit 5 est toujours fixé sur les minuscules et le fait d'annuler ce bit revient à convertir la lettre en sa majuscule). Pressez la touche "x" de votre clavier pour interrompre le programme.

```
s$="Mot de test"
FOR i%=1 TO LEN(s$)
  PRINT CHR$(BCHG(ASC(MID$(s$,i%)),5));
NEXT i%
```

--> Affiche 'mOT DE TEST' sur l'écran. Chaque minuscule est donc convertie en majuscule et chaque majuscule en minuscule. Cette méthode ne s'applique cependant pas aux accents.

SHL(x,y)	**SHL&**(x,y)	**SHL|**(x,y)
SHR(x,y)	**SHR&**(x,y)	**SHR|**(x,y)
ROL(x,y)	**ROL&**(x,y)	**ROL|**(x,y)
ROR(x,y)	**ROR&**(x,y)	**ROR|**(x,y)

```
(x,y : iexp)
```

Ces instructions décalent (SHift) ou font subir une rotation (ROtate) au contenu d'une expression numérique x sur y bits. Si aucun type de variable précis n'est spécifié, l'opération s'effectue sur une longueur de long mot (4 octets), si vous spécifiez '&' sur une longueur de mot (2 octets) et si vous spécifiez un '|' sur une longueur d'octet. La troisième lettre du nom de fonction définit le sens du décalage ou de la rotation. Dans ce cas, 'L' signifie vers la gauche (Left) et R vers la droite (Right).

Pour les fonctions de mot (&), le bit 15 est en outre copié dans les bits 16 à 31 et pour les fonctions d'octet (|), les bits 8 à 31 sont annulés.

Les exemples numériques suivants illustrent l'effet des instructions *Shift*.

```
x%    SHL|(x,1)   BIN$(x%,16)          BIN$(SHL|(x%,1),16)
18    36          00000000 00010010    00000000 00100100
642   4           00000010 10000010    00000000 00000100

x%    SHL&(x%,1)  BIN$(x%,16)          BIN$(SHL&(x%,1),16)
18    36          00000000 00010010    00000000 00100100
130   4           00000000 10000010    00000001 00000100

x%    SHR&(x%,2)  BIN$(x%,16)          BIN$(SHR&(x%,2),16)
24    6           00000000 00011000    00000000 00000110
4162  1040        00010000 01000010    00000100 00010000
```

(Les bits ont été groupés par huit uniquement par souci de clarté, BIN$ n'effectue pas de tel regroupement).

Les exemples numériques suivants concernent les instructions de rotation. Ces instructions réinsèrent du côté opposé les bits qui sortent du domaine numérique utilisé.

Supposons par exemple que seul le bit le plus élevé d'un octet soit fixé. Cet octet subit maintenant une rotation d'un bit sur la gauche (ROL|(128,1)). Le bit expulsé sur la gauche est maintenant réintroduit à droite, de sorte que le premier bit du résultat de la fonction est mis. Avec SHL|(128,1), le bit mis expulsé aurait simplement été annulé.

Voici d'autres exemples numériques :

```
x|    ROL|(x,1)    BIN$(x|,8)    BIN$(ROL|(x|,1),8)
6     12           00000110      00001100
130   5            10000010      00000101

x|    ROR|(x|,3)   BIN$(x|,8)    BIN$(ROR|(x|,3),8)
66    144          01000010      00000000
2     64           00000010      01000000
```

Exemple :

```
x|=128+1                    !fixe les bits 7 et 0
x|=ROR|(x|,1)               !y est fixé sur 192
PRINT SHL(y%,4),y%*2^4
PRINT SHL(ROR|(128+1,1),4)
```

--> Le nombre 3072 apparaît trois fois sur le moniteur. La fonction SHL(a,b) équivaut à l'expression a*2^b tant qu'aucun bit n'est expulsé du domaine de définition de quatre octets. La formulation avec la fonction de bit est toutefois nettement plus rapide.

AND(x,y)
OR(x,y)
XOR(x,y)
IMP(x,y)
EQV(x,y)

```
(x,y : iexp)
```

Ces fonctions sont des combinaisons logiques de deux expressions numériques. Dans le résultat de la fonction AND, seuls sont mis les bits qui sont mis aussi bien dans x que dans y. Le résultat de la fonction OR comporte des bits mis dans les positions dans lesquelles un bit est mis dans x ou dans y ou encore dans les deux. XOR ne fixe que les bits qui sont fixés soit dans x, soit dans y mais pas dans les deux à la fois. Autrement dit XOR fixe les bits dont les valeurs diffèrent dans x et y.

IMP n'affecte un zéro à un bit que si le bit correspondant est mis dans x mais non dans y, sinon le bit est mis dans le résultat de la fonction. EQV fixe un bit si les bits correspondants ont les mêmes valeurs dans x et y. Les tables de vérité présentées dans la section sur les opérateurs logiques peuvent vous aider à bien comprendre le fonctionnement de ces instructions.

Exemples :

```
x=3
y=2
z=AND(x,y)  !z devient 2
PRINT OR(2,7),XOR(x,1+4+8)
```

--> Sur l'écran apparaissent les nombres 7 et 14.

```
PRINT BIN$(15,4),15
PRINT BIN$(6,4),6
PRINT BIN$(IMP(15,6),4),"IMP(15,6)"
PRINT BIN$(EQV(15,6),4),"EQV(15,6)"
```

--> Sur l'écran apparaissent :

```
1111    15
0110    6
0110    IMP(15,6)
0110    EQV(15,6)
```

SWAP(x)

```
x : iexp
```

La fonction SWAP interprète l'expression numérique x comme un long mot (4 octets) dont elle échange les mots supérieur et inférieur (de 2 octets chacun). Cette fonction n'a rien à voir avec l'instruction GFA-BASIC homonyme. Elle est employée dans un certain nombre de situations bien précises (par exemple pour transmettre un paramètre de long mot en deux mots à une routine du système d'exploitation ou pour traiter un pointeur avec un ordre de mots inversé).

Exemple :

```
x=1044480
PRINT BIN$(x,32)
y=SWAP(x)
PRINT BIN$(y,32)
```

--> Sur le moniteur apparaît :

```
00000000000011111111000000000000
11110000000000000000000000001111
```

Voici un exemple d'application :

```
~WIND_SET(0,13,SWAP(t%),t%,0,0)
```

BYTE(x)
CARD(x)
WORD(x)

```
(x : iexp)
```

BYTE renvoie les 8 bits inférieurs de l'expression numérique x. CARD lit de même les 16 bits inférieurs de x. WORD étend un mot à un long mot (c'est-à-dire que le bit 15 est copié dans les bits 16 à 31).

Exemple :

```
PRINT BYTE(1+254),BYTE(1+255)
PRINT HEX$(CARD(&H1234ABCD))
```

--> Ecrit 255, 0 et ABCD sur l'écran.

5. GESTION DES CHAINES DE CARACTERES

(LEFT$, RIGHT$)
(MID$ en tant que fonction)
(PRED, SUCC)
(LEN)
(INSTR, RINSTR)
(STRING$, SPACE$, SPC)
(UPPER$)
(LSET, RSET, MID$ en tant qu'instruction)

Les instructions LEFT$ et RIGHT$ renvoient la partie gauche ou droite d'une chaîne de caractères. MID$ permet, en tant que fonction, d'extraire des parties situées au milieu d'une chaîne de caractères. Lorsque MID$ est utilisée comme instruction, elle permet d'insérer une chaîne à l'intérieur d'une autre. PRED et SUCC renvoient le caractère de code ASCII inférieur ou supérieur d'une unité à l'expression de chaîne spécifiée.

LEN détermine la longueur d'une chaîne de caractères, INSTR et RINSTR recherchent une chaîne à l'intérieur d'une autre chaîne de caractères. STRING$, SPACE$ et SPC servent à produire des chaînes de caractères contenant plusieurs fois la même expression de chaîne. UPPER$ convertit tous les caractères d'une chaîne en majuscules. LSET et RSET permettent d'insérer une chaîne dans une autre en l'alignant sur la gauche ou la droite de cette chaîne.

**LEFT$**(a$ [,x])
**RIGHT$**(a$ [,x])

```
(a$ : sexp)
(x : iexp)
```

LEFT$ renvoie les x premiers caractères de la chaîne de caractères a$. Si x est supérieur au nombre de caractères de a$, a$ est renvoyée intégralement. Si x n'est pas précisé, seul le premier caractère de la chaîne a$ est renvoyé. RIGHT$ renvoie de même les x derniers caractères de a$. Si x n'est pas indiqué, c'est le dernier caractère de a$ qui est transmis.

Exemples :

```
a$="Manuel de BASIC"
b$=LEFT$("GFA-MICRO APPLICATION",4)
PRINT b$;RIGHT$(a$,5)
```

--> Le mot 'GFA-BASIC' sera écrit sur le moniteur.

```
a$="Olivier"
b$=LEFT$(a$)+RIGHT$("Pasternak")
PRINT b$
```

--> 'Ok' apparaît sur l'écran.

MID$(a$,x [,y]) en tant que fonction

```
(a$ : sexp)
(x,y : iexp)
```

La fonction MID$ renvoie y caractères à partir de la position x de la chaîne de caractères a$. Si x est supérieur à la longueur de a$, une chaîne de caractères vide est renvoyée. Si y est omis, c'est tout le reste de la chaîne à partir du x-ième caractère qui sera renvoyé.

Exemple :

```
a$="Manuel de GFA-BASIC"
b$=MID$(a$,11,9)+MID$("Version 3.0",8)
PRINT b$
```

--> Le texte 'GFA-BASIC 3.0' apparaît sur l'écran.

PRED(a$)
SUCC(a$)

```
(a$ : sexp)
```

PRED renvoie le caractère de code ASCII immédiatement inférieur à celui de l'expression de chaîne spécifiée. Seul le premier caractère d'une chaîne de caractères est donc évalué. SUCC renvoie de même le caractère immédiatement supérieur. PRED(a$) équivaut donc à l'expression CHR$(PRED(ASC(a$))) et SUCC(a$) à CHR$(SUCC(ASC(a$))).

Ces deux fonctions peuvent également être appliquées aux expressions entières (voyez la section consacrée à l'arithmétique entière).

Exemple :

```
caractere$="B"
predecesseur$=PRED(caractere$)
successeur$=SUCC(caractere$)
PRINT predecesseur$,caractere$,successeur$
```

--> Affiche les lettres A,B et C sur l'écran.

LEN(a$)
TRIM$(a$)

```
a$ : sexp
```

LEN détermine combien de caractères comporte la chaîne de caractères a$ et renvoie ce nombre.

TRIM$ élimine les espaces aux extrémités gauche et droite d'une chaîne de caractères.

Exemples :

```
a$="Test"
x=LEN(a$)+1
PRINT x,LEN("Mot")
```

--> Affichage des nombres 5 et 3.

```
b$="  test  "
PRINT len(b$)
PRINT TRIM$(b$)
PRINT LEN(TRIM$(b$))
```

--> 8, la chaîne "test" sans espaces et 4 apparaissent.

INSTR(a$,b$)
INSTR(a$,b$, [x])
INSTR([x] ,a$,b$)

```
a$,b$ : sexp
x : iexp
```

La fonction INSTR recherche la chaîne de caractères b$ à l'intérieur de la chaîne de caractères a$. Si x est précisé, la recherche commence à partir du x-ième caractère de a$, sinon la recherche commence à partir du premier caractère. La position à partir de laquelle b$ figure dans a$ est renvoyée. Si b$ n'a pas été trouvée, INSTR renverra zéro. Si a$ et b$ sont des chaînes vides, INSTR renverra un.

Exemples :

```
a$="MICRO APPLICATION"
x=INSTR(a$,"appli")
PRINT x,INSTR("GFA-BASIC","BASIC",6)
```

--> Les nombres 7 et 0 apparaissent sur l'écran.

RINSTR(a$,b$)
RINSTR(a$,b$, [x])
RINSTR([x] ,a$,b$)

```
(a$,b$ : sexp)
(x : iexp)
```

RINSTR() recherche une chaîne dans une autre, comme INSTR, mais en commençant la recherche à partir de la fin de la chaîne de caractères.

Exemple :

```
PRINT RINSTR("A:\DOSSIER\*.GFA","\")
```

--> recherche le dernier '\' dans le nom de chemin et affiche la position du caractère trouvé (11 en l'occurrence).

**STRING$**(x,a$)
STRING$(x,code)
SPACE$(x)
SPC(x)

```
x : iexp
a$ : sexp
```

La fonction STRING$ produit une chaîne de caractères contenant x fois (entre 0 et 32767) l'expression de chaîne 'a$' ou la valeur ASCII 'code'. Si le paramètre 'code' est spécifié, c'est naturellement CHR$(code) qui sera employé.

SPACE$ produit une chaîne contenant x espaces. SPC affiche x espaces dans une instruction PRINT. Contrairement à SPACE$, SPC ne produit pas de chaîne de caractères qui puisse par exemple être affectée à une variable à l'aide d'un signe égal.

Exemple :

```
a$="GFA "
b$=SPACE$(5)
PRINT b$;STRING$(3,a$);SPC(4);STRING$(5,"<")
```

--> Sur l'écran apparaît ' GFA GFA GFA <<<<<'

**UPPER$**(a$)

```
(a$ : sexp)
```

Convertit en majuscules toutes les minuscules d'une chaîne de caractères. Cela fonctionne également pour les accents.

Exemple :

```
a$="Texte"
b$=UPPER$(a$)+UPPER$(" de "+"test de")
PRINT b$;UPPER$(" Gfa-Basic 3.0")
```

--> Sur l'écran apparaît 'TEXTE DE TEST DE GFA-BASIC 3.0'.

LSET a$=b$
RSET a$=b$
MID$(a$,x [,y]) en tant qu'instruction

```
(a$ : svar)
(b$ : sexp)
(x,y : iexp)
```

LSET et RSET insèrent l'expression de chaîne b$ dans la variable de chaîne a$ en l'alignant sur la gauche (LSET) ou sur la droite (RSET), en la comblant d'espaces le cas échéant de sorte que la longueur de a$ n'est pas modifiée.

En tant qu'instruction, MID$ sert à insérer une expression de chaîne au milieu d'une variable de chaîne. Par exemple :

```
MID$(a$,x,y)=b$
```

insèrera l'expression de chaîne b$ dans la variable a$ à partir du x-ième caractère. Le paramètre optionnel y définit combien de caractères de b$ peuvent être insérés au maximum dans a$. Si y est omis, un nombre de caractères aussi grand que possible sera repris dans a$. MID$ ne modifie pas la longueur de a$ car les caractères venant de b$ remplacent simplement les parties correspondantes de a$. Si a$ est trop courte pour recevoir b$ entièrement, l'insertion est interrompue.

Exemples :

```
a$="      "
FOR i%=1 TO 128          ! Sort des colonnes de nombres
  LSET a$=STR$(i%)       ! alignées sur la gauche
  PRINT a$;
NEXT i%
PRINT
FOR i%=1 TO 128
  RSET a$=STR$(i%)       ! et alignées sur la droite
  PRINT a$;
NEXT i%
~INP(2)
```

--> affiche des colonnes de nombres formatées alignées sur la gauche et sur la droite. a$=STR$(i%,5,0) reviendrait au même que RSET a$=STR$(i%,5,0). Pour sortir du programme, il vous suffit de presser une touche quelconque du clavier, grâce à "~INP(2)".

```
a$="GF   ASIC"
MID$(a$,3)="A-B"
b$="Mot de test"
MID$(b$,8,4)="qui était-ce ?"
PRINT a$,b$
```

--> Les textes 'GFA-BASIC' et 'Mot de qui' apparaissent.

6. ENTREE AU CLAVIER ET SORTIE SUR ECRAN

Ce chapitre vous présente un certain nombre de possibilités de base pour l'entrée et la sortie. Il commence par le test d'un caractère du clavier avec INKEY$. Vient ensuite le test d'une variable à l'aide de INPUT et des instructions apparentées LINE INPUT, FORM INPUT et FORM INPUT AS.

L'exposé des possibilités de sortie commence par l'instruction la plus simple, PRINT. Ensuite seront présentées les versions étendues de cette instruction (PRINT AT, PRINT USING), puis les instructions de test (CRSCOL, CRSLIN, POS) ou de définition (TAB, HTAB, VTAB) de la position du curseur.

Ce chapitre se conclura par la présentation des instructions KEYxxx. Ces nouvelles instructions permettent une manipulation très simple, pendant le déroulement du programme, des fonctions liées au clavier.

ENTREE AU CLAVIER

INKEY$

INKEY$ identifie un caractère entré à l'aide du clavier. Cette fonction ne permet cependant pas de tester les touches de commutation du clavier *(Shift, Alternate, Control, CapsLock)*. INKEY$ n'attend pas qu'une touche quelconque ait été actionnée mais renvoie une chaîne vide si aucune touche n'a été actionnée depuis le dernier test du clavier. Dans le cas contraire, la fonction renvoie le caractère ASCII correspondant à la touche actionnée. Si cependant la touche actionnée ne possède pas de code ASCII, par exemple s'il s'agit d'une touche de fonction, de la touche *Help* ou de la touche *Undo*, c'est le code clavier de la touche actionnée qui est renvoyé. Dans ce cas, une chaîne de deux caractères est renvoyée dont le premier caractère est CHR$(0) et le second le code de la touche spéciale actionnée. L'exemple suivant permet de déterminer les valeurs obtenues avec *Inkey$*.

Exemple :

```
DO
  t$=INKEY$
  IF t$<>""
    IF LEN(t$)=1
      PRINT "Code ASCII :  ";ASC(t$),"Caractère ASCII :  ";t$
    ELSE
      PRINT CHR$(0), "Code clavier : ";CVI(t$)
    ENDIF
  ENDIF
LOOP
```

--> Affiche le code ASCII ou bien le code clavier de chaque touche actionnée. Remarque : CVI(t$) équivaut ici à ASC(RIGHT$(t$)) puisque ASC(t$)=0. Pour interrompre le programme, pressez simultanément les touches CONTROL, SHIFT et ALTERNATE.

INPUT ["texte",] x [,y,...]
INPUT ["texte";] x [,y,...]

```
(x,y : avar ou svar)
```

L'instruction INPUT peut être utilisée avec différentes variantes. Elle sert à entrer des variables ou des listes de variables avec ou sans explications préliminaires ("texte").

INPUT part toujours de la dernière position du curseur. PRINT AT suivi d'un point-virgule ou LOCATE, VTAB, HTAB permettent de placer le curseur de façon à définir l'emplacement où l'entrée doit être effectuée.

Lorsque le mot d'instruction INPUT est suivi d'un texte, ce texte peut être séparé des variables suivantes par une virgule ou par un point-virgule. Si vous utilisez un point-virgule, un point d'interrogation et un espace seront ajoutés au texte et le curseur sera placé à la suite de l'ensemble. Si vous utilisez une virgule, le curseur marquant l'entrée sera placé immédiatement à la suite du dernier caractère du texte explicatif.

Si INPUT n'est suivi d'aucun texte, un point d'interrogation apparaîtra dans tous les cas, suivi d'un espace puis du curseur.

Si une seule variable doit être définie, la fin de l'entrée de cette variable doit être confirmée en actionnant la touche *Return* ou *Enter*. Si plusieurs variables doivent être définies avec une seule instruction INPUT, chaque variable peut être confirmée en actionnant *Return* ou *Enter* ou bien les variables peuvent aussi être séparées par des virgules et entrées en appuyant une seule fois sur la touche *Return* ou *Enter*.

Pour que des virgules puissent toutefois être entrées comme faisant partie d'une chaîne lue avec INPUT, il faut utiliser LINE INPUT.

Lorsque c'est une variable numérique qui est attendue mais que l'utilisateur entre un texte, un signal d'erreur retentit et l'entrée doit être recommencée. Tant que la touche *Return* ou *Enter* n'a pas été actionnée, des caractères peuvent être effacés de l'entrée à l'aide des touches *Backspace* et *Delete*. Les touches *curseur gauche* et *droite* peuvent également être employées. En appuyant sur la touche *Insert*, on peut commuter entre les modes d'insertion et d'effacement. La longueur maximale pour une entrée est de 255 caractères.

Les caractères spéciaux peuvent être entrés de trois manières différentes :

- En appuyant sur <ALTERNATE> et sur les chiffres du bloc numérique, par exemple : tenez la touche *Alternate* enfoncée et tapez le nombre 64 sur le bloc de touches numériques. Lorsque vous relâcherez la touche *Alternate*, @ apparaîtra. Cela vaut aussi pour INKEY$, INP(2), les dialogues GEM, etc..., tant que cette possibilité n'a pas été désactivée avec KEYPAD.

- En entrant <CONTROL> <S> et un autre caractère, par exemple : <CONTROL> <S> <c> pour le caractère Pi.

- En entrant <CONTROL> <A> puis le code ASCII du caractère voulu (suivant KEYPAD, voyez à cet endroit, par exemple : <CONTROL> <A> <2> <7> pour le caractère *Escape*).

Exemple :

```
INPUT a$
INPUT "",b$
INPUT "Deux nombres s'il vous plaît : ";x,y
PRINT a$,b$,x,y
```

--> Entre deux chaînes et deux variables numériques. La première attente se présente avec '? ', la seconde sans texte et la troisième avec le message *'Deux nombres s'il vous plaît : ?'*.

LINE INPUT ["texte",] a$ [,b$...]
LINE INPUT ["texte";] a$ [,b$...]

```
(a$,b$ : svar)
```

LINE INPUT est une variante de l'instruction INPUT. Contrairement à INPUT, elle permet d'entrer des virgules comme partie d'une variable de chaîne.

Pour le reste, les explications données pour INPUT en ce qui concerne l'entrée de variables ou d'une liste de variables, la correction de l'entrée jusqu'à ce que la touche *Return* ou *Enter* ait été actionnée et les autres variantes possibles s'appliquent également ici. Cette instruction ne fonctionne toutefois qu'avec les variables de chaînes.

Exemple :

```
LINE INPUT a$
INPUT b$
PRINT a$,b$
```

--> Veuillez entrer deux fois le texte *'Vir,gule'*. Vous verrez ensuite apparaître sur le moniteur *'Vir,gule'* et *'Vir'*. Voyez aussi LINE INPUT #.

FORM INPUT n,a$
FORM INPUT n AS a$

```
(n : iexp)
(a$ : svar)
```

FORM INPUT et FORM INPUT AS servent toutes deux à l'entrée de variables de chaînes. n indique ici le nombre de caractères (entre 1 et 255) que la chaîne a$ devra comporter au maximum.

FORM INPUT AS affiche en outre la valeur actuelle de a$ qui peut ainsi être éditée par l'utilisateur. Les possibilités d'édition pour ces deux instructions sont les mêmes que pour l'instruction INPUT.

Exemple :

```
FORM INPUT 10,a$
b$="test"
FORM INPUT 5 AS b$
PRINT a$,b$
```

--> Réclame deux chaînes. Lors de l'entrée de la seconde chaîne, le mot *'test'* est proposé pour l'édition en tant que valeur prédéfinie de b$.

PRINT
PRINT expression
PRINT AT(colonne,ligne);expression
WRITE expression
LOCATE ligne,colonne

```
(expression : n'importe quelle sexp ou aexp ou combinaison des deux)
(colonne,ligne : iexp)
```

L'instruction PRINT sans paramètre déclenche un saut de ligne. Si le curseur se trouve déjà sur la dernière ligne de l'écran, l'écran est entièrement décalé d'une ligne vers le haut. PRINT suivie d'une expression sert à sortir cette expression dans la position actuelle du curseur. Les chaînes de caractères doivent être placées entre guillemets. Si l'expression à sortir se compose de plusieurs éléments (constantes, variables ou expressions), les différentes parties peuvent être séparées par des points-virgules, virgules ou apostrophes.

L'emploi de virgules permet de placer le curseur après la prochaine position de colonne divisible par 16. Si la dernière colonne est atteinte, le curseur saute à la ligne suivante. Le point-virgule permet de sortir les éléments spécifiés sans le moindre intervalle. Avec l'apostrophe, un espace sera inséré entre les différents éléments.

PRINT AT permet de positionner l'expression à sortir sur une colonne et une ligne déterminées. Suivant la résolution utilisée vous pouvez disposer de 80 colonnes et 25 lignes au maximum. L'emploi de fenêtres restreint cependant les limites fixées au positionnement du curseur.

Lorsque l'expression à sortir n'est pas terminée par un point-virgule, le curseur est placé au début de la ligne suivante. S'il se trouvait déjà sur la dernière ligne, l'écran est décalé d'une ligne vers le haut.

Lorsque des caractères de contrôle (codes jusqu'à 31) sont spécifiés avec PRINT, ces codes sont traités par l'émulateur VT-52 (voyez l'annexe).

Contrairement à PRINT AT, LOCATE sert uniquement à positionner le curseur sur la position de colonne et de ligne spécifiée. Il n'est donc pas possible d'indiquer des expressions à sortir à la suite de LOCATE (voyez VTAB/HTAB).

L'instruction WRITE sert à stocker des données dans des fichiers séquentiels de telle façon qu'elles puissent ensuite être relues avec INPUT. L'instruction WRITE est suivie d'expressions numériques et d'expressions de chaînes séparées entre elles par des virgules.

Lors de la sortie, les expressions seront séparées par des virgules et les expressions de chaînes seront placées entre guillemets. Remarque : lorsqu'il s'agit d'une sortie sur disquette, c'est ce format qui convient le mieux à une relecture ultérieure avec INPUT. Un point-virgule peut aussi figurer à la suite de la dernière expression d'une instruction WRITE, auquel cas la sortie ne se terminera pas par l'envoi de CR/LF.

Exemples :

```
a$="GFA-MICRO APPLICATION"
PRINT Left$(a$,4)+"BASIC"'1+2;
PRINT ".0","GFA-";UPPER$(MID$(a$,5))
```

--> Sur l'écran apparaît le texte 'GFA-BASIC 3.0' et 'GFA-MICRO APPLICATION'.

```
PRINT AT(4,8);"sur la quatrième colonne de la huitième ligne"
```

--> écrit une chaîne sur la position 4,8.

```
LOCATE 8,4
PRINT "sur la huitième ligne, en quatrième colonne"
```

--> positionne le curseur sur la quatrième colonne de la huitième ligne, puis sort une chaîne dans cette position avec l'instruction PRINT qui suit.

```
WRITE 1+1,"Salut",3*4
```

--> affiche 2,"Salut",12.

PRINT USING format$,expression,[;]
PRINT AT(ligne,colonne);USING format$,expression [;]

```
format$ : sexp
expression : un nombre illimité de sexp ou aexp séparées par des virgules
colonne,ligne : iexp
```

PRINT USING et sa variante PRINT AT USING servent à produire une sortie de données formatée sur l'écran. Ces deux instructions fonctionnent pour l'essentiel comme PRINT et PRINT AT si ce n'est que les données figurant dans l'expression à sortir sont formatées d'après le contenu de format$.

Les symboles suivants permettent de définir le format des expressions numériques :

\# Emplacement réservé pour un chiffre. Si c'est le dernier chiffre de l'instruction de formatage, un arrondissement sera effectué lors de la sortie.

. Sert à séparer les chiffres décimaux parmi plusieurs caractères #.

, Insère une virgule dans la position correspondante entre les caractères # de façon à opérer une séparation entre les chiffres de milliers.

- Le signe *moins* ne peut figurer qu'en première ou dernière position de la chaîne de format. Il réserve un emplacement pour la sortie d'un signe négatif. Si le signe est positif, un espace sera sorti dans cet emplacement.

+ Comme avec le signe *moins*, un signe *plus* précèdera les nombres positifs. Les signes *moins* et *plus* ne peuvent être combinés dans la chaîne de format.

* Remplace #. En cas de zéros au début d'un nombre, remplace les espaces symbolisant normalement ces zéros par des caractères *étoile*.

$ Fait afficher un signe $ avant un nombre si ce symbole est placé immédiatement avant le premier symbole #.

^ Sert à fixer le format exponentiel (E+000). Les signes # déterminent dans ce cas la longueur de la mantisse (y compris les caractères E+ ou E-). Si plusieurs # figurent avant le point décimal, l'exposant est adapté de façon à être divisible par ce nombre de #. Pour les nombres négatifs, un signe doit absolument être prévu.

Les symboles suivants permettent de définir le format des chaînes de caractères :

& Fait sortir la chaîne de caractères intégralement.

! Limite la sortie au premier caractère de la chaîne.

\...\ Indique combien de caractères d'une chaîne de caractères doivent être sortis (y compris les deux caractères \).

_ Le caractère *souligné* fait sortir le caractère suivant dans la spécification de format.

Il est en outre possible de faire sortir des *inserts* de texte non modifiés entre ces indications de format. Les variables, listes de variables ou expressions suivant la chaîne de format sont séparées par des virgules.

Exemples :

```
PRINT USING "#.####",PI
PRINT AT(4,4); USING "PI_._._. #.###",PI;
```

--> Les textes '3.1416' et 'PI... 3.142' apparaissent sur l'écran.

```
FOR i%=1 TO 14
  PRINT USING "###.##^^^^ ",2^i%;
NEXT i%
```

--> affichera :

1.00E+00	2.00E+00	4.00E+00	8.00E+00	16.00E+00
32.00E+00	64.00E+00	128.00E+00	256.00E+00	512.00E+00
1.02E+03	2.05E+03	4.10E+03	8.19E+03	16.38E+03

MODE permet d'échanger points et virgules.

MODE n

```
n : iexp
```

MODE permet de choisir entre le point décimal et la virgule des milliers et la virgule décimale et le point des milliers. MODE permet aussi de sélectionner le format d'affichage de la date. Le point et la virgule s'appliquent à PRINT USING et STR$(x,v,n). Le réglage de la date vaut pour DATE$, SETTIME, DATE$= et FILES.

Le paramètre n peut recevoir des valeurs entre 0 et 3, pour définir un des modes présentés dans la table suivante :

Paramètre n	USING	DATE$
MODE 0	#,###.##	16.05.1988
MODE 1	#,###.##	05/16/1988
MODE 2	#.###,##	16.05.1988
MODE 3	#.###,##	05/16/1988

DEFNUM n

```
(n : iexp)
```

DEFNUM modifie la sortie des nombres par l'instruction PRINT et ses variantes. Toutes les sorties de nombres après une instruction DEFNUM s'effectueront avec n chiffres (avant et après la virgule, le point n'étant pas pris en compte). La précision de calcul interne n'est pas affectée. Lors de ces sorties, le chiffre numéro n+1 sera pris en compte pour l'arrondissement.

Exemple :

```
PRINT 100/3
DEFNUM 5
PRINT 100/3
```

--> Fait apparaître les nombres 33.333333333 et 33.333 sur l'écran.

CRSCOL
CRSLIN
POS(x)
TAB(colonne)
HTAB colonne
VTAB ligne

```
n,colonne,ligne : iexp
x : aexp
```

Le groupe d'instructions CRSCOL, CRSLIN, POS et TAB sert à tester et à fixer la position du curseur. CRSCOL fournit la position de colonne actuelle du curseur et CRSLIN sa position de ligne actuelle. POS fournit le nombre de caractères sortis sur l'écran depuis le dernier retour de chariot (Carriage Return) AND 255. L'expression x ne sert à rien et est purement et simplement ignorée.

La valeur renvoyée par POS ne coïncide pas nécessairement avec la position actuelle du curseur. Si une chaîne de caractères de 120 caractères est sortie, par exemple, le curseur figurera à la colonne 40 mais POS renverra la valeur 120. Surtout pour la sortie de caractères de commande, POS(0) a peu de rapport avec la colonne du curseur car seuls les caractères sont comptés. LF (CHR$(10)) n'est donc pas pris en compte, un CR (CHR$(13)) fixe le compteur sur zéro et un BS (CHR$(8)) diminue le compteur d'une unité.

TAB(n) sort le nombre d'espaces nécessaires pour que POS(0) atteigne la valeur de n. Si n a déjà été dépassé, un saut de ligne (CR/LF) est tout d'abord effectué, comme avec PRINT sans paramètre. Ici aussi, seul AND 255 est utilisé.

Les instructions HTAB et VTAB amènent le curseur sur une position de colonne ou de ligne déterminée.

Pour les positionnements du curseur, le coin supérieur gauche a la coordonnée 1.

Exemples :

```
PRINT AT(38,12);"Test ";
PRINT CRSCOL'CRSLIN
PRINT Tab(37);"Test ";
PRINT POS(0)
~INP(2)
```

--> Sur l'écran apparaît, au milieu, 'Test 43 12' et 'Test 42'.

```
PRINT AT(4,3);"Mot 1"
HTAB 4
VTAB 2
PRINT "Mot 2"
```

--> Sort le texte 'Mot 1' en quatrième colonne de la troisième ligne et le texte 'Mot 2' en quatrième colonne de la seconde ligne.

LES INSTRUCTIONS KEYXXX

Ce nouveau groupe d'instructions permet de contrôler l'état des touches de commutation du clavier pendant le déroulement du programme ainsi que la définition des chaînes de caractères (de 31 caractères au maximum) qui peuvent librement être attribuées aux touches de fonction et qui sont également disponibles sous l'éditeur GFA-BASIC.

KEYPAD n

```
(n : iexp)
```

L'expression numérique n est évaluée bit par bit et revêt la signification suivante :

Bit	Signification	0	1
0	NUM-LOCK	désactivé	activé
1	NUM-LOCK	ne peut être activé	peut être activé
2	CTRL-KEYPAD	normal	curseur
3	ALT-KEYPAD	normal	entrée ASCII
4	KEYDEF sans ALT	désactivé	activé
5	KEYDEF avec ALT	désactivé	activé

Exemples :

```
PRINT "Veuillez appuyer sur <Esc>"
REPEAT                    ! parcourt la boucle
UNTIL INKEY$=CHR$(27)     ! jusqu'à ce que la touche Esc soit
'                           actionnée
PRINT "Veuillez à nouveau appuyer sur <Esc>"
'
REPEAT                    ! fait de même
  KEYTEST n|              ! mais différemment
UNTIL n|=27
```

--> Attend deux fois que la touche *Escape* soit actionnée.

```
PRINT "Veuillez actionner une touche"
frapp_1|=INP(2)             ! Attendre qu'une touche soit actionnée
PRINT "Veuillez à nouveau actionner une touche"
KEYGET frapp_2|             ! La même chose à nouveau
PRINT "INP(2) : ";frapp_1|  ! et sortir les valeurs ASCII
PRINT "KEYGET : ";frapp_2|  ! des touches actionnées
```

--> Attend deux fois qu'une touche soit actionnée et sort les codes des touches actionnées.

```
DO
  KEYGET a%
  PRINT HEX$(a%,8)'BIN$(a%,32)'
  OUT 5,a%
  PRINT
LOOP
```

--> Attend qu'une touche soit actionnée, sort le code correspondant en hexadécimal et en binaire et affiche le caractère. Remarque : OUT 5 entraîne, de même que VID:, un affichage des caractères de commande (<32).

KEYPRESS n

```
n : iexp
```

L'instruction KEYPRESS simule le fait qu'une touche soit actionnée. A cet effet, l'expression numérique n doit recevoir la valeur ASCII de la touche qui doit être simulée. Dans le mot fort peuvent en outre être spécifiés le code clavier et l'état des touches de commutation du clavier (cf. KEYGET).

Exemple :

```
FOR i&=65 TO 90      ! simule le fait d'appuyer
  KEYPRESS i&        ! sur les touches des majuscules
NEXT i&              ! A à Z
'
KEYPRESS 13          ! simule le fait d'appuyer
'                    ! sur la touche Return
LINE INPUT a$        ! lit des caractères jusqu'au premier retour
'                    ! de chariot, c'est-à-dire CHR$(13) ou la touche
PRINT a$             ! Return, les place dans la chaine a$ et les
'                    ! sort

KEYPRESS &H3B0000    ! appuie sur la touche de fonction F1
KEYPRESS &1001B      ! appuie sur la touche Esc de telle façon que
                     ! les entrées dans les sélecteurs d'objets gérés
                     ! par GEM puissent également fonctionner.

KEYDEF 1,"Salut"+CHR$(13)
KEYPRESS &H83B0000
PAUSE 1
LINE INPUT a$
PRINT a$
```

--> Définit un texte qui devra être sorti lorsqu'on appuiera sur *Alternate-F1*, simule ensuite le fait d'appuyer sur ces deux touches puis attend de façon à ce que la routine d'interruption appropriée ait le temps de traiter la définition des touches et entre enfin le "Salut" par INPUT.

KEYDEF n,s$

```
n : iexp
s$ : sexp
```

L'instruction KEYDEF permet de définir les touches de fonction en leur affectant des chaînes de caractères quelconques d'une longueur maximale de 31 caractères. A cet effet, une expression arithmétique valant entre 1 et 20 est spécifiée à la suite de l'instruction KEYDEF. Les valeurs de 1 à 10 de n permettent d'appeler les touches de fonction F1 à F10, les valeurs 11 à 20 de n permettant de spécifier des valeurs pour la touche Shift et les touches de fonction F1 à F10.

La définition des touches de fonction ainsi opérée s'applique pendant le déroulement du programme ainsi que sous l'éditeur. Sous l'éditeur GFA-BASIC, cette définition ne s'applique toutefois que lorsque la touche *Alternate* est actionnée en plus des touches indiquées (cf. KEYPAD bits 4 et 5).

Exemple :

```
KEYDEF 1,"F1"
KEYDEF 11,"Shift+F1"
```

--> Sort le texte 'F1' lorsque vous appuyez sur la touche de fonction F1 et le texte 'Shift+F1' lorsque vous appuyez sur les touches Shift et F1 (chaque fois en liaison avec *Alternate*).

ENTREES ET SORTIES GENERALES

Le présent chapitre explique tout d'abord la lecture de constantes (DATA, READ, RESTORE). Ensuite est abordée la gestion des répertoires (DIR$, CHDIR, DIR, FILES, MKDIR, RMDIR). La structure du système hiérarchisé de fichiers est expliquée lors de la description de ces instructions.

Nous nous intéressons ensuite à l'ouverture, à la fermeture et au changement de nom des fichiers (EXIST, KILL, NAME, OPEN, CLOSE). Nous vous présentons encore les possibilités d'entrée et de sortie de zones de mémoire (BLOAD, BSAVE, BGET, BPUT).

Nous présentons alors les possibilités d'accès séquentielles (INPUT$, INPUT#, PRINT#) et séquentielles indexées (SEEK, RELSEEK) ainsi que les fichiers d'accès relatif (FIELD, GET#, PUT#, SEEK#, RELSEEK#). La section consacrée à la périphérie expose les possibilités d'entrée et de sortie octet par octet offertes par INP, OUT ainsi que les fonctions de test correspondantes INP?, OUT?. Elle décrit également la réception à travers l'interface sérielle et l'interface Midi (INPAUX$, INPMID$).

Le chapitre se conclut par le test de la souris et du joystick (MOUSE, MOUSEX, MOUSEY, MOUSEK, HIDEM, SHOWM, STICK, STRIG) ainsi que par la sortie sur imprimante (LPRINT, LPOS, HARDCOPY).

LIGNES DE DONNEES

DATA const [,const1,const2,...]
READ var [,var1,var2,...]
RESTORE [mar]

```
(const,const1,const2 : constantes numériques ou de chaine)
(var,var1,var2 : avar ou svar)
(mar : nom défini par l'utilisateur)
```

DATA permet de spécifier des valeurs constantes sans grande consommation de place. Ces données sont ensuite lues avec READ. Les valeurs numériques peuvent être spécifiées à cet effet sous forme hexadécimale, octale ou binaire. Lorsque c'est une chaîne de caractères contenant des virgules qui doit être lue, l'ensemble de la chaîne doit être placé entre guillemets.

Les instructions DATA et READ fonctionnent avec ce qu'on appelle un pointeur de données. Ce pointeur désigne en permanence la prochaine valeur DATA qui devra être lue avec READ. Au début du programme, ce pointeur désigne systématiquement la première valeur après le premier DATA.

L'instruction RESTORE permet de rediriger ce pointeur de données vers des lignes de DATA bien précises. A cet effet, on place devant la ligne DATA voulue une marque *'mar'* sur laquelle le pointeur de données sera ensuite dirigé avec RESTORE *mar*. Si l'instruction RESTORE n'est suivie d'aucune marque, le pointeur de données est dirigé sur la première valeur DATA du programme.

La séquence de caractères constituant la marque *mar* peut être composée de chiffres, lettres, caractères souligné et de points. Contrairement aux noms de variables, elle peut aussi commencer par un chiffre. Lors de sa définition, elle doit être terminée par un double point.

Exemple :

```
FOR i=1 TO 3
  READ a
  PRINT a'
NEXT i
'
RESTORE Chiffres_romains
READ a$,b$,c$,d$
PRINT
PRINT a$'b$'c$'d$
'
DATA 1,2,3,4
DATA a,b,c,d
'
Chiffres_romains:
DATA I,II,III,IV
```

--> Les valeurs numériques 1, 2 et 3 sont lues dans une boucle et affichées sur l'écran. Le pointeur de données est ensuite dirigé à l'aide de RESTORE sur la ligne DATA comportant les chiffres romains. Cette redirection s'opère à l'aide de la marque *Chiffres_romains*. Ces données sont ensuite affectées aux variables de chaîne a$,b$,c$ et d$, puis affichées sur l'écran.

```
DATA 10,&A,$A,&HA,&O12,&X1010,%1010
FOR i%=1 TO 7
  READ a%
  PRINT a%
NEXT i%
```

--> Lit sept fois le nombre 10. Avec READ/INPUT/VAL, etc..., les nombres hexadécimaux peuvent aussi être désignés par $ et les nombres binaires par % et non seulement par &H, & ou &X.

GESTION DE FICHIERS

Cette section explique les instructions servant à l'organisation des fichiers. La compréhension de ces instructions suppose toutefois que soit connue la structure des spécifications de fichier telle qu'elle est régie par les règles du système de fichiers hiérarchisé. Une spécification de fichier se compose de trois parties : la désignation du lecteur, le nom de fichier et la marque de fichier. La désignation du lecteur comporte la marque du lecteur A à O, suivie d'un double point. Le nom de fichier comporte de 1 à 8 caractères. Une marque de fichier peut également être indiquée en option. Elle se compose d'un point suivi de 1 à 3 caractères.

Pour classer les fichiers, on utilise également des répertoires (qu'on appelle aussi *dossiers*). Ces répertoires peuvent être créés sur différents niveaux. Le niveau le plus bas est appelé *répertoire racine*. De ce *répertoire racine* procède une ramification en sous-répertoires. Une entrée de répertoire peut donc se composer de 3 parties :

- Désignation du lecteur
- Nom(s) du (des) sous-répertoire(s)
- Nom et marque du fichier

Ces parties sont séparées par des traits de fraction renversés "\" *(Backslash)*. Les noms des répertoires présentent le même format que les noms de fichiers. Le chemin d'accès à un fichier est donc obtenu en accolant les éléments présentés ci-dessus. En premier lieu, la désignation du lecteur, suivie des noms de sous-répertoires, puis du nom de fichier, par exemple :

```
A:\TEXTE.DOC\MANUEL\CHAP_1.DOC
```

Dans cet exemple, le chemin d'accès se compose donc des éléments :

```
A:              Désignation du lecteur
\TEXTE.DOC      Répertoire TEXTE.DOC
\MANUEL         Sous-répertoire MANUEL
\CHAP_1         Nom de fichier CHAP_1
.DOC            Marque de fichier .DOC
```

Vous pouvez utiliser en outre deux caractères spéciaux qui facilitent la sélection du fichier.

Ces caractères spéciaux peuvent être placés à l'intérieur du nom de fichier et de sa marque. Il s'agit du point d'interrogation et de l'étoile (le signe de multiplication en informatique).

Le point d'interrogation désigne n'importe quel caractère valable dont le code ASCII soit supérieur à 32. Il est également possible de placer le point d'interrogation en plusieurs endroits, y compris consécutifs, de la spécification de fichier. L'étoile signifie que n'importe quelle séquence de caractères peut combler les positions suivantes. Ces caractères sont aussi appelés *"jockers"*.

REPERTOIRES

DFREE(n)
CHDRIVE n ou n$
DIR$(n)
CHDIR nom$

```
n : iexp
nom$ : sexp
```

DFREE (disk free) fournit la place mémoire libre, en octets, sur un lecteur de disquette. L'exécution de cette fonction peut être longue, surtout pour les partitions de disque dur.

CHDRIVE (change drive) fixe le lecteur standard. Si aucun lecteur particulier n'est spécifié dans une instruction d'entrée ou de sortie, l'entrée ou la sortie s'effectue à partir de ou sur le lecteur standard. Au lieu du numéro de lecteur sous forme de nombre, une chaîne peut aussi être spécifiée pour CHDRIVE. Dans ce cas, le premier caractère désigne la lettre-code du lecteur.

DIR$(n) détermine le chemin d'accès actuel sur un lecteur tel qu'il a été fixé avec CHDIR.

Pour DFREE(n), DIR$(N) et CHDRIVE, la variable n contient la marque du lecteur. n peut revêtir toute valeur de 0 à 16, 0 correspondant au lecteur standard et 1 à 16 aux lecteurs A à P.

CHDIR fixe le répertoire actuel. Comme CHDIR ne permet pas de changer de lecteur, cette instruction s'applique toujours au lecteur actuel ou au lecteur spécifié. CHDIR "B:\TEST" changera donc le dossier par défaut pour le lecteur B:, c'est-à-dire le dossier qui sera appelé lors des accès à ce lecteur sans indication de chemin, c'est-à-dire sans \. L'expression de chaîne nom$ contient ici le chemin d'accès souhaité. Si nom$ ne contient qu'un trait de fraction renversé "\", c'est le répertoire *racine* du lecteur actuel qui est sélectionné.

Il existe encore deux noms de dossier spéciaux, "." et "..". Le nom de dossier "." représente une autre possibilité pour désigner le dossier activé et ".." de même une autre possibilité pour désigner le dossier immédiatement supérieur dans la hiérarchie.

Lorsque les dossiers \TEST\A1 et \TEST\A2 figurent par exemple sur une disquette et que le dossier \TEST\A1 est activé (avec CHDIR), CHDIR "..\A2" permettra de passer au dossier \TEST\A2.

Exemples :

```
CHDRIVE 1
PRINT DFREE(0)
PRINT DIR$(2)
CHDRIVE "C:\"
```

--> CHDRIVE 1 fixe le lecteur A comme lecteur standard. PRINT DFREE(0) sort sur l'écran la place mémoire libre (en octets) sur le lecteur standard (n=0. Maintenant, à cause de CHDRIVE 1, le lecteur A:). PRINT DIR$(2) détermine le chemin d'accès sur le lecteur B:. Le lecteur C est ensuite fixé comme lecteur standard.

```
CHDIR "\"
CHDIR "TEXTE.DOC\MANUEL"
CHDIR "ANNEXE"
```

--> CHDIR "\" sélectionne le répertoire racine du lecteur actuel. La seconde ligne sélectionne le fichier MANUEL du répertoire TEXTE.DOC. La troisième ligne sélectionne alors le sous-répertoire ANNEXE (\TEXTE.DOC\MANUEL\ANNEXE)

DIR p$ [TO nom$]
FILES p$ [TO nom$]

```
(p$,nom$ : sexp)
```

Les instructions DIR et FILES servent à afficher des répertoires. Lorsque DIR est employé, le répertoire d'un lecteur déterminé est sorti sur l'écran suivant le format standard. Le chemin d'accès voulu doit être indiqué à cet effet dans l'expression de chaîne p$. Il en va de même pour l'instruction FILES mais celle-ci présente en outre la date, l'heure et la longueur des fichiers. D'autre part, des noms de dossiers précédés de * seront également sortis s'ils correspondent au masque de recherche (par exemple *.*), y compris dans certains cas les dossiers "." et "..". Si le masque de fichier p$ se termine par ":" ou "\", GFA-BASIC ajoutera de lui-même "*.*" (cf. FSFIRST et FSNEXT).

Pour DIR ou FILES, il est possible, en option, d'ajouter le suffixe TO nom$, qui permet d'envoyer le répertoire dans un fichier ou sur un périphérique. L'expression de chaîne nom$ contient dans ce cas le nom d'un fichier ou une désignation de périphérique.

Exemples :

```
DIR "A:\TEXTES\*.DOC"
DIR "A:\TEXTES\MANUEL\*.DOC" TO "B:\MANUEL\TABLE.ASC"
DIR "A:\*.*" TO "PRT:"
FILES "A:\*.DOC" TO "LST:"
```

--> La première ligne sort sur l'écran tous les noms de fichier portant la marque ".DOC" (dans le dossier \TEXTES sur le lecteur A). La seconde ligne envoie le répertoire dans un fichier "TABLE.ASC" sur le lecteur B. La sortie sur imprimante est réalisée par la troisième ligne. La quatrième ligne envoie sur l'imprimante tous les noms de fichiers dotés de la marque ".DOC".

FGETDTA()
FSETDTA(adr)

```
adr : iexp
```

La fonction FGETDTA() renvoie l'adresse de la DTA (disk transfer area).

FSETDTA fixe la DTA. Un sélecteur de fichier modifiera également la DTA. La DTA est au départ fixée sur BASEPAGE+128. Cette adresse est également utilisée par DIR, FILES et EXIST.

La DTA présente la structure suivante :

Octets	Offset	Signification
21	0	Réservé à GEMDOS
1	21	Attributs de fichier (voir plus bas)
2	22	Heure
2	24	Date
4	26	Longueur du fichier
14	30	Nom de fichier terminé par un octet nul, sans espaces.

La signification des bits d'attribut est la suivante :

Bit	Signification
0	Fichier protégé contre l'écriture.
1	Fichier caché.
2	Fichier système.
3	Nom de disquette.
4	Dossier.
5	Bit d'archivage.

FSFIRST(p$,attr)
FSNEXT()

```
p$ : sexp
attr : iexp
```

La fonction FSFIRST permet de faire rechercher le premier fichier correspondant au critère de recherche spécifié dans p$ (par exemple C:*.GFA). Les noms de fichiers identifiés ainsi que d'autres informations sont écrits dans la DTA. Le paramètre *attr* contient les attributs que peuvent comporter les fichiers recherchés.

La fonction FSNEXT() recherche le prochain fichier remplissant les conditions définies pour FSFIRST.

Exemple :

```
~FSETDTA(BASEPAGE+128)              ! Fixe la DTA
e%=FSFIRST("\*.GFA",-1)             ! Fixer le critère de recherche
DO UNTIL e%
  IF BYTE(BASEPAGE+149) AND 16      ! Si c'est un dossier,
    PRINT "*";CHAR(BASEPAGE+158),   ! placer une étoile devant le
  ELSE                              ! nom de fichier, sinon
    PRINT ' CHAR(BASEPAGE+158),     ! un espace
  ENDIF
  e%=FSNEXT()
LOOP
```

--> Sort sur l'écran tous les fichiers portant l'extension .GFA dans le répertoire actuel, ainsi que, marqués par *, tous les noms de dossiers possédant également cette extension.

MKDIR nom$
RMDIR nom$

```
(nom$ : sexp)
```

L'instruction MKDIR (make directory) crée un répertoire (dossier). L'expression de chaîne *nom$* contient le chemin d'accès correspondant. RMDIR (remove directory) supprime un répertoire (dossier) à la condition que ce répertoire ne contienne ni sous-répertoires ni fichiers.

Exemple :

```
MKDIR "A:\TEXTES"
RMDIR "A:\TEXTES"
```

--> La première ligne crée un répertoire "TEXTES" sur le lecteur A. La seconde ligne supprime ce répertoire.

FICHIERS

EXIST(nom$)

```
nom$ : sexp
```

EXIST permet de déterminer si un fichier existe. L'expression de chaîne *nom$* doit contenir à cet effet le chemin d'accès à ce fichier. La fonction renvoie TRUE (-1) si le fichier existe, sinon FALSE (0) (cf. FSFIRST et FSNEXT).

Exemple :

```
OPEN "U",#1,"TEST.TXT"
PRINT #1,"EXEMPLE"
CLOSE #1
PRINT EXIST("TEST.TXT")
PRINT EXIST("TEST.DOC")
```

--> Les trois premières lignes ouvrent un fichier appelé "TEST.TXT". La quatrième ligne teste si le fichier "TEST.TXT" existe. La réponse obtenue est TRUE. Dans la dernière ligne, le test aboutit à la réponse FALSE car le fichier n'existe pas. Voyez aussi FSFIRST et FSNEXT.

OPEN mode$, #n,nom$ [nmb]

```
mode$,nom$ : sexp
nmb,n : iexp
```

OPEN ouvre un canal de données vers un fichier ou un périphérique. L'expression de chaîne *"mode$"* définit une des possibilités d'accès suivantes :

O (output)	Ouvre un fichier en écriture. A cet effet, le fichier est créé s'il n'existait pas déjà ou bien ses données sont supprimées s'il existait déjà.
I (input)	Ouvre un fichier en lecture.
A (append)	Permet d'ajouter des données à la suite d'un fichier existant. Le pointeur de données est à cet effet dirigé sur la fin du fichier.
U (update)	Ouvre un fichier déjà existant en lecture et écriture.
R (random access)	Ouvre un fichier relatif (fichier à accès sélectif) en lecture et écriture. Ce type de fichiers est décrit sous l'instruction FIELD.

L'expression numérique "n" contient le numéro de canal qui est une valeur quelconque entre 0 et 99. Ce numéro de canal doit être indiqué pour travailler avec le fichier. Le caractère # devant la désignation du canal peut être omis. L'expression de chaîne *"nom$"* contient le chemin d'accès du fichier. Une désignation de périphérique (voir liste ci-dessous) peut également être indiquée comme nom de fichier. L'expression numérique "nmb" n'est indiquée que pour les fichiers en accès sélectif (random access) ; elle contient dans ce cas la longueur d'un enregistrement.

Abréviation		Signification	Utilisation interne
LST:	(list, printer)	Imprimante	BIOS 0
AUX:	(auxial)	Sériel (RS 232)	BIOS 1
CON:	(console)	Clavier	BIOS 2 ou VDI
MID:	(musical instruments digital interface)	Interface MIDI	
IKB:	(intelligent keyboard)	Processeur clavier	BIOS 4
VID:	(video)	Moniteur	BIOS 5 ou VDI
PRN:	(printer)	Imprimante	GEMDOS -3

LOF(#n)
LOC(#n)
EOF(#n)
CLOSE [#n]
TOUCH [#] n

```
nmb, n : iexp
```

Les fonctions LOF (length of file), LOC (location) et EOF (end of file) ne peuvent être appliquées qu'à des fichiers préalablement ouverts avec OPEN. Les trois fonctions ont en commun l'expression numérique "n". Cette expression indique le numéro de canal du fichier auquel s'applique la fonction. LOF fournit la longueur d'un fichier en octets. LOC indique la position actuelle du pointeur de données, toujours en octets (voyez aussi SEEK). EOF indique si le pointeur de fichier est dirigé sur la fin d'un fichier (c'est-à-dire si un fichier a été entièrement lu). Lorsque le pointeur de fichier désigne la fin du fichier, TRUE (-1) est renvoyé, sinon FALSE (0).

CLOSE referme un canal de données vers un fichier ou un périphérique préalablement ouvert avec OPEN. L'expression numérique "n" contient ici le numéro du canal à fermer. Si aucun canal n'est spécifié, tous les fichiers ouverts sont refermés.

TOUCH actualise l'indication de l'heure d'un fichier, c'est-à-dire que les entrées de date et d'heure d'un fichier ouvert sont fixées sur les données actuelles correspondantes du système.

Exemples :

```
OPEN "O",#1,"test.txt"
FOR i%=1 TO 20
  PRINT #1,STR$(i%)
NEXT i%
CLOSE #1
FILES "test.txt"
DELAY 20                        ! Pause de 20 secondes
OPEN "U",#1,"test.txt"
TOUCH #1
CLOSE #1
FILES "test.txt
```

--> Cet exemple ouvre le fichier *"test.txt"* en écriture sous le numéro de canal 1, puis le referme. Les caractéristiques de ce fichier sont ensuite sorties, actualisées après une pause de 20 secondes puis à nouveau sorties.

```
OPEN "I",#1,"test.txt"
PRINT "   Longueur du fichier : ";LOF(#1)
PRINT
PRINT "   Données","Position du pointeur de données"
DO UNTIL EOF(#1)
  INPUT #1,a$
  PRINT "   ";a$,LOC(#1)
LOOP
CLOSE #1
```

--> Cet exemple ouvre le même fichier en lecture. La longueur du fichier est sortie à l'aide de LOF. Le contenu du fichier est ensuite sorti sur l'écran avec la position correspondante du pointeur de fichier. La condition d'interruption de la boucle est fixée avec EOF. Les données sont lues en vérifiant le pointeur de données (quelle que soit la longueur de fichier calculée précédemment). La boucle est donc parcourue jusqu'à ce que la fonction EOF renvoie TRUE (-1) comme résultat.

NAME ancien$ **AS** nouveau$
RENAME ancien$ **AS** nouveau$
KILL nom$

```
(ancien$,nouveau$,nom$ : sexp)
```

NAME renomme un fichier dont le chemin d'accès était *ancien$* en un fichier avec *nouveau$* pour chemin d'accès. Le contenu du fichier n'est pas modifié. Les fichiers *ancien$* et *nouveau$* doivent figurer sur le même lecteur. RENAME équivaut à l'instruction NAME.

KILL supprime un fichier dont le chemin d'accès est défini par l'expression de chaîne *"nom$"*.

Exemples :

```
OPEN "O",#1,"test.txt"
PRINT #1,"Exemple"
CLOSE #1
'
NAME "test.txt" AS "exemple.txt"
PRINT "Répertoire avant KILL :"
DIR
PRINT "Répertoire après KILL :"
KILL "exemple.txt"
DIR
```

--> Dans le premier exemple, le fichier "test.txt" est ouvert en écriture. Ce fichier est ensuite renommé "exemple.txt", puis finalement supprimé. Cette suppression est vérifiée par un examen du répertoire. Après NAME et KILL, les répertoires modifiés sont affichés sur l'écran.

BLOAD n$ [,adr]
BSAVE n$,adr,nmb
BGET #n,adr,nmb
BPUT #n,adr,nmb

```
(nom$ : sexp)
(n,adr,nmb : aexp)
```

BSAVE permet de sauvegarder sur disquette (ou disque RAM, disque dur, etc...) une zone de mémoire que l'on pourra ensuite recharger avec BLOAD, par exemple. L'expression numérique *adr* indique l'adresse de départ de la zone de mémoire. Si *adr* n'est pas spécifiée, c'est la dernière adresse spécifiée avec BSAVE qui sera utilisée. Pour BSAVE, il faut indiquer en outre *nmb* (la longueur du fichier n$). Le paramètre n$ représente le nom du fichier à charger ou à sauvegarder. Pour n$ s'appliquent les règles du système hiérarchisé de fichiers décrites sous DIR.

BSAVE, BLOAD permettent seulement d'appeler, sous le nom approprié, des fichiers entiers. BPUT et BGET permettent par contre d'accéder à des fichiers à travers leur numéro de canal. BGET et BPUT permettent aussi de charger ou de sauvegarder des parties de fichier (même à plusieurs reprises).

Exemples :

```
DEFFILL 1,2,4
PBOX 100,100,200,200
BSAVE "rectangl.pic",XBIOS(2),32000
CLS
PRINT AT(4,20);"Image sauvegardée, veuillez frapper une touche !"
~ INP(2)
CLS
BLOAD "rectangl.pic"
```

--> dessine un rectangle et sauvegarde l'écran sous "rectangl.pic". Ensuite apparaît un message approprié. Dès qu'une touche est frappée, le fichier sauvegardé est rechargé à l'écran.

```
DEFFILL 1,2,4
PBOX 0,0,639,199
DEFFILL 1,2,2
PBOX 0,200,639,399
```

```
DEFTEXT 1,0,0,32
TEXT 100,115,"La moitié supérieure"
TEXT 100,315,"La moitié inférieure"
'
OPEN "O",#1,"screen.pic"
BPUT #1,XBIOS(2),32000
CLOSE #1
PAUSE 25
'
OPEN "I",#1,"screen.pic"
BGET #1,XBIOS(2)+16000,16000
BGET #1,XBIOS(2),16000
CLOSE #1
```

--> Remplit la moitié supérieure de l'écran avec un motif et la moitié inférieure de l'écran avec un autre. L'écran entier est ensuite sauvegardé dans le fichier *screen.pic* puis rechargé après une pause. La première moitié du fichier est toutefois chargée dans la moitié inférieure de l'écran, la seconde dans la moitié supérieure de l'écran.

ACCES SEQUENTIEL

INP(#n)
OUT #n,a [,b,c,...]

```
n,a,b,c : iexp
```

INP(#n) lit un octet dans un fichier précédemment ouvert. OUT envoie de même un octet dans un fichier. L'expression numérique "n" contient ici le numéro de canal (0 à 99) sous lequel le fichier voulu est appelé. INP et OUT sans "#" peuvent également être utilisés pour la communication avec l'écran, IKBD, l'interface MIDI, etc... (voir plus bas).

OUT utilise BYTE(a).

Exemple :

```
OPEN "O",#1,"test.txt"
OUT #1,128
CLOSE #1
'
OPEN "I",#1,"test.txt"
a=INP(#1)
```

```
CLOSE #1
PRINT a
```

--> Dans la première partie de l'exemple, un fichier est ouvert en écriture et un octet y est écrit qui sera relu dans la seconde partie pour être chargé dans la variable numérique "a" et être sorti sur l'écran. "a" a ici la valeur 128.

INPUT$(nmb [,#n])

```
n, nmb : iexp
```

INPUT$ lit au clavier une chaîne de caractères composée de nmb caractères. L'indication d'un numéro de canal "n" (0 à 99), optionnelle, permet de lire des caractères dans un fichier. Dans les deux cas, l'expression numérique "nmb" indique combien de caractères doivent être lus.

Exemples :

```
OPEN "O",#1,"version.dat"
PRINT #1,"GFA-BASIC, Version 3.0"
CLOSE #1
'
OPEN "I",#1,"version.dat"
v$=INPUT$(9,#1)
CLOSE #1
PRINT v$
PRINT "Veuillez indiquer le numéro de version : ";
PRINT INPUT$(3)
```

--> Dans le premier exemple, le fichier "version.dat" est ouvert et un message est écrit dans le fichier. La seconde partie de l'exemple lit les 9 premiers caractères de ce fichier pour les placer dans la variable de chaîne v$ puis les sort sur l'écran. Ensuite apparaît un message puis 3 caractères sont lus au clavier puis sortis sur l'écran.

INPUT #n,var1 [,var2,var3,...]
LINE INPUT #n,a1$ [,a2$,a2$,...]

```
n : iexp
a1$, a2$, a3$ : sexp
var1, var2, var3 : avar ou svar
```

INPUT #n permet de charger des données à partir d'un fichier.

Il est possible de lire des valeurs une à une ou au contraire des listes de variables, les différentes variables étant dans ce cas séparées entre elles par des virgules. Ces deux instructions sont décrites plus précisément sous INPUT et LINE INPUT, la seule différence étant que la lecture ne se fait pas ici (en général) sur le clavier.

Exemple :

```
OPEN "I",#1,"TEXT.DOC"
INPUT #1,a$,b$
LINE INPUT #1,c$
CLOSE #1
PRINT a$
PRINT b$
PRINT c$
```

--> charge trois chaînes à partir d'un fichier (qui doit avoir été ouvert préalablement) et les sort sur l'écran.

PRINT #n,expression
PRINT #n,**USING** format$,expression
WRITE #n,expression

```
(n : iexp)
(format$ : sexp)
(expression : aexp ou sexp ou une combinaison des deux)
```

PRINT #n sort des données sur un canal de données. PRINT #n, USING permet une sortie formatée sur un canal de données. Dans les deux cas, l'expression numérique "n" représente le numéro de canal (0 à 99) du fichier voulu. Ces deux instructions sont décrites plus précisément sous PRINT, PRINT USING et WRITE. Notez toutefois que PRINT #n,AT(,) n'est pas possible.

L'instruction WRITE sert essentiellement à gagner de la place mémoire lors du stockage des données dans des fichiers séquentiels. Les diverses expressions sont séparées par des virgules et les chaînes de caractères doivent être placées entre parenthèses.

Exemples :

```
OPEN "O",#1,"TEXT.DOC"
b$="mot"
PRINT #1,"Test",a$
PRINT #1,"GFA-","BASIC"
CLOSE #1
```

--> écrit trois chaînes dans un fichier (qui doit avoir été créé préalablement) et les sort sur l'écran.

```
OPEN "O",#1,"TEST.DAT"
WRITE #1,"Version ",3,".0"
CLOSE #1
'
OPEN "I",#&,"TEST.DAT"
INPUT #1,v1$,v2$,v3$
CLOSE #1
'
PRINT v1$+v2$+v3$
```

--> Ecrit dans le fichier TEST.DAT des données séparées par des virgules, relit ensuite les données avec INPUT puis les sort sur l'écran.

STORE #i,x$() [,n]
RECALL #i,x$(),n,x

```
i,n : iexp
x$() : tableau de chaînes
x : Variable d'au moins 32 bits
```

L'instruction STORE sert à sauvegarder un tableau de chaînes de caractères sous forme d'un fichier de texte (avec CR/LF comme séparation). Le tableau de chaînes entier est sorti sur le canal i ouvert. Le paramètre optionnel n permet de spécifier combien d'éléments du tableau de chaînes doivent être sortis.

L'instruction RECALL sert à charger rapidement un tableau de chaînes à partir d'un fichier de texte. n lignes du fichier de texte sont chargées dans le tableau de chaînes. Si n est trop grand pour le dimensionnement du tableau de chaînes, le nombre de lignes à charger est automatiquement limité (n = -1 charge le tableau tout entier). Si la fin du fichier (EOF) est atteinte lors du chargement, le chargement s'achève sans message d'erreur. Dans tous les cas, la variable x contient à la fin de l'opération le nombre de chaînes de caractères effectivement chargées.

Exemple :

```
DIM A$(1000)
FOR i%=0 TO 499
  a$(i%)=STR$(RND)     !N'importe quoi
NEXT i%
OPEN "O",#1,"test.txt"
STORE #1,a$(),500
CLOSE #1
```

```
'
DIM B$(2000)
OPEN "I",#1,"test.txt"
RECALL #1,b$(),-1,n
CLOSE #1
PRINT n
```

--> Le nombre de lignes chargées est affiché, soit 500 ici.

```
PRINT "Compteur de lignes"
DIM a$(1000)
DO
  FILESELECT "\*.*","",f$
  EXIT IF f$=""
  lc%=0
  OPEN "I",#1,f$
  DO
    RECALL #1,a$(),-1,x%
    ADD lc%,x%
  LOOP WHILE x%
  CLOSE #1
  PRINT f$;" contient ";lc%;" lignes"
LOOP
```

--> Ce programme compte les lignes des fichiers de texte.

Remarque : STORE fonctionne également avec les fichiers orientés caractère (AUX:) mais pas RECALL qui utilise un SEEK en interne.

SEEK #n,pos
RELSEEK #n,nmb

```
(n,nmb,pos : iexp)
```

Les instructions SEEK et RELSEEK servent à positionner le pointeur de données, ce qui permet de réaliser un accès séquentiel indexé au fichier. L'expression numérique "n" contient ici le numéro de canal d'un fichier préalablement ouvert avec OPEN. Les deux instructions ne peuvent être utilisées que pour des fichiers mais pas pour des périphériques. Le pointeur de données indique quel octet d'un fichier a été lu ou écrit en dernier. Le pointeur de données vaut automatiquement 0 lors de l'ouverture d'un fichier, sauf en mode "A" (Append = ajouter des données à un fichier existant). Les instructions de lecture et écriture commencent par l'octet désigné par le pointeur de données.

L'instruction SEEK permet un positionnement absolu du pointeur de données. Le pointeur de données est dirigé à cet effet sur l'octet indiqué par "pos". L'instruction RELSEEK sert par contre au positionnement relatif du pointeur, le pointeur de données étant décalé de la valeur indiquée par "nmb" par rapport à sa position actuelle. RELSEEK sera généralement plus rapide !

Les expressions numériques "nmb" et "pos" ne peuvent revêtir que des valeurs comprises entre 0 et la longueur du fichier. Les valeurs positives déplacent le pointeur de données vers la fin du fichier, les valeurs négatives le déplacent vers le début du fichier. Si la valeur 0 est spécifiée, le pointeur de données est fixé sur le début du fichier.

Exemple :

```
OPEN "O",#1,"X.X"
PRINT #1,STRING$(20,42)
SEEK #1,10
PRINT #1,"#";
RELSEEK #1,-5
OUT #1,33,48
CLOSE #1
OPEN "I",#1,"X.X"
LINE INPUT #1,a$
PRINT a$
CLOSE #1
```

--> Affichage produit : ******!0**#*********

ACCES SELECTIF

Cette section sera consacrée à la manipulation des fichiers relatifs ou fichiers d'accès sélectif (Random Access). Pour ce type de fichiers, deux notions jouent un rôle particulièrement important, la notion d'enregistrement et la notion de champ. Un enregistrement représente un groupe logique de données. Si l'on crée par exemple un fichier d'adresses, chaque enregistrement représentera une adresse, c'est-à-dire l'ensemble des renseignements, nom, prénom, rue, ville, etc... qui seront regroupés pour chacune des personnes figurant dans ce fichier d'adresses. L'enregistrement se décompose en champs, par exemple "nom", "rue". De ces notions découlent les notions de longueur d'enregistrement et de longueur de champ. La longueur d'enregistrement est le nombre total d'octets pour chaque enregistrement, autrement dit la somme des longueurs des champs composant un enregistrement.

La différence fondamentale entre un fichier relatif (fichier d'accès sélectif) et un fichier séquentiel réside dans l'accès aux données.

Avec un fichier séquentiel, le fichier doit toujours être entièrement chargé dans la mémoire pour qu'il soit possible d'accéder à un enregistrement quelconque. Avec un fichier relatif, par contre, un enregistrement peut être lu sans qu'il soit nécessaire pour cela de charger le fichier tout entier, ce qui est naturellement très intéressant dès lors qu'on a affaire à de très longs fichiers. Cet avantage en mémoire est cependant compensé par un inconvénient sur la disquette puisqu'un fichier relatif occupe nettement plus de place sur la disquette du fait qu'il travaille avec des longueurs de champs et d'enregistrement fixes. Ces longueurs d'enregistrement et de champs sont en effet respectées pour la sauvegarde de toutes les données, quelle que soit la longueur réelle de ces données.

FIELD #n,nmb AS enreg$ [,nmb AS enreg$, nmb AS enreg$,...]
FIELD #n,nmb AT(x) [,nmb AT(x) [,nmb AT(x) [,,...]

```
n, nmb,: iexp
enreg$: svar, mais pas une variable de tableau
```

L'instruction FIELD AS subdivise les enregistrements en champs. L'expression numérique "n" est le numéro du canal de données (0 à 99) d'un fichier ouvert avec OPEN. L'expression numérique "nmb" fixe la longueur de champ. La variable de chaîne enreg$ reçoit un champ d'un enregistrement. Si l'enregistrement doit être subdivisé en plusieurs champs, les différentes parties de l'instruction (nmb AS enreg$) doivent être séparées par des virgules. La somme des différentes longueurs de champ doit être égale à la longueur d'enregistrement, faute de quoi un message d'erreur approprié sera sorti. Pour que les différents enregistrements de données n'aient pas une longueur différente de celle fixée par l'instruction FIELD, il est conseillé d'utiliser les instructions LSET et RSET ou MID$.

AT permet par exemple d'écrire des variables numériques dans un fichier R (Random Access, c'est-à-dire fichier relatif) sans que celles-ci aient à être transférées au préalable dans des chaînes de caractères. Un pointeur sur la variable numérique à stocker doit pour cela être placé entre parenthèses et le nombre d'octets qui devront être lus ou écrits à partir de cette adresse doit être inscrit devant AT. N'importe quelles autres adresses peuvent également être spécifiées. Par exemple :

```
FIELD #1,4 AT(*a%),2 AT(*b&),8 AT(*c#)
```

AS et AT peuvent aussi être librement combinés, par exemple :

```
FIELD #2,4 AS a$,2 AT(*b&),8 AT(*c#),6 AS d$
```

Contrairement à ce qui était le cas dans la version 2.0, une instruction FIELD peut être écrite sur plusieurs lignes. La longueur maximale d'enregistrement est de 32767, le nombre maximal de champs de 500 environ.

GET #n, [,enregistrement]
PUT #n, [,enregistrement]
RECORD #n,enregistrement

```
n,enregistrement : iexp
```

GET lit un enregistrement quelconque dans un fichier relatif (fichier "R"). PUT écrit de même un enregistrement quelconque dans un fichier relatif. L'expression numérique "n" représente ici le numéro (0 à 99) d'un canal de données ouvert avec OPEN. Le paramètre optionnel "enregistrement" est une valeur comprise entre 1 et le nombre d'enregistrements (65535 au maximum) du fichier traité. Cette valeur fixe le numéro de l'enregistrement à lire ou à écrire. A défaut d'indication de "enregistrement", c'est l'enregistrement suivant qui est lu ou écrit.

RECORD fixe simplement le prochain numéro d'enregistrement pour PUT ou GET. Après RECORD #1,15, GET #1, lira donc l'enregistrement numéro 15.

Attention : un fichier ne peut jamais être agrandi que d'un seul enregistrement à la fois ou bien de plusieurs enregistrements si ces enregistrements sont mis en place à l'aide d'une boucle.

Exemples :

```
OPEN "R",#1,"\prive.rdm",65
FIELD #1,24 AS nom$,24 AS rue$, 5 AT(*codepostal%), 12 AS ville$
'
FOR i%=1 TO 3
  INPUT "Nom     : ",n$
  INPUT "Rue     : ",r$
  INPUT "Code p.: ",codepostal%
  INPUT "Ville   : ",v$
  LSET nom$=n$
  LSET rue$=r$
  LSET ville$=v$
  PUT #1,i%
  CLS
NEXT i%
'
CLOSE #1
```

--> Un fichier relatif (mode "R") est ouvert avec une longueur d'enregistrement de 65 octets. A l'aide de l'instruction FIELD, un enregistrement sera subdivisé en 2 champs de 24 octets, un champ de 5 octets et un champ de 12 octets. La somme des champs (24+24+5+12) donne bien 65 (c'est-à-dire la longueur d'enregistrement spécifiée avec l'instruction OPEN).

Les données d'adresses sont ensuite entrées et placées dans les variables correspondantes alignées sur la gauche. L'enregistrement entier est ensuite écrit dans le fichier.

```
OPEN "R",#1,"A:\prive.rdm",65
FIELD #1,24 AS nom$,24 AS rue$, 5 AT (*codepostal%), 12 AS ville$
FOR i%=1 TO 3
  GET #1,i%
  PRINT "N° enregistr.: ";i%
  PRINT "Nom          : ";nom$
  PRINT "Rue          : ";rue$
  PRINT "Code postal  : ";codepostal%
  PRINT "Ville        : ";ville$
NEXT i%
'
CLOSE #1
```

--> Le fichier relatif (d'accès sélectif) "PRIVE.RDM" est ouvert et 3 enregistrements sont lus.

COMMUNICATION AVEC LA PERIPHERIE

Entrée et sortie octet par octet

INP(n)
INP?(n)
OUT [#]n,a [,b...]
OUT?(n)

---

n,a,b : iexp

INP lit un octet sur la périphérie. L'expression numérique "n" peut revêtir des valeurs de 0 à 5 (voir table ci-dessous) ou contenir un numéro de canal. L'instruction OUT envoie des octets à la périphérie. Contrairement à ce qui était le cas sous les versions GFA-BASIC 2.xx, OUT permet maintenant d'envoyer plusieurs octets à la périphérie ou sur un fichier.

INP? ou OUT? déterminent respectivement l'état d'entrée ou de sortie d'un périphérique. Un nombre différent de zéro est renvoyé en réponse si le périphérique indiqué sous n est prêt, sinon FALSE (0).

n	Abréviation		Signification
0	LST:	(list, printer)	Imprimante
1	AUX:	(auxial)	sériel (RS 232)
2	CON:	(console)	Clavier
3	MID:	(musical instruments digital interface)	Interface MIDI
4	IKB:	(intelligent keyboard)	Processeur clavier
5	VID:	(video)	Moniteur

Exemples :

```
PRINT AT(4,4);"Veuillez frapper une touche !"
~ INP(2)
```

--> sort un message sur l'écran et attend qu'une touche soit actionnée (numéro de périphérique 2, clavier).

```
OUT 2,27,69,13
```

--> équivaut à l'instruction CLS, vide l'écran et fixe le curseur dans le coin supérieur gauche à l'aide de la séquence VT52 Clear Screen (esc + "E").

INTERFACES SERIELLE ET MIDI

INPAUX$
INPMID$

INPAUX$ et INPMID$ permettent de lire très rapidement des chaînes à partir des interfaces sérielle et MIDI.

Exemples :

```
DO
  PRINT INPAUX$;
LOOP UNTIL MOUSEK
```

--> lit très rapidement toutes les données du buffer *Input* de l'interface sérielle. Ce type d'entrée des données est beaucoup plus rapide qu'une lecture octet par octet sur l'interface.

```
inp_aux$ = ""
WHILE INP?(1)
  inp_aux$ = inp_aux$ + CHR$(INP(1))
WEND
```

--> Cette variante lit les données beaucoup plus lentement.

SOURIS ET JOYSTICK

MOUSEX
MOUSEY
MOUSEK
MOUSE mx,my,mk

```
(mx,my,mk : avar)
```

Les instructions MOUSEX, MOUSEY et MOUSEK servent à tester la position actuelle de la souris ainsi que l'état des boutons de la souris. L'instruction MOUSE permet de regrouper tous ces tests. MOUSE renvoie en effet les coordonnées actuelles de la souris dans mx,my et l'état des boutons de la souris dans mk. La variable mk peut revêtir des valeurs comprises entre 0 et 3.

mk	Bouton actionné
0	aucun
1	gauche
2	droit
3	gauche et droit

Exemple :

```
REPEAT
  IF MOUSEK=1
    PLOT MOUSEX,MOUSEY
  ENDIF
UNTIL MOUSEK=2
'
REPEAT
  MOUSE mx%,my%,mk%
  IF mk%=2
    PLOT mx%,my%
  ENDIF
UNTIL mk%=1
```

--> La boucle REPEAT teste la position et l'état des boutons de la souris. Un point est dessiné dans l'emplacement mx,my uniquement lorsque le bouton gauche est tenu enfoncé. La boucle est abandonnée dès que le bouton droit de la souris est actionné. Dans la seconde boucle, le bouton droit permet de dessiner et le bouton gauche de quitter le programme.

SETMOUSE mx,my

```
mx,my,mk : iexp)
```

L'instruction SETMOUSE permet un positionnement du curseur de la souris géré par programme. Le paramètre optionnel mk peut simuler le fait d'appuyer sur ou le fait de relâcher les boutons de la souris. Cela ne vaut malheureusement que pour la VDI et non (ou rarement) pour l'AES.

Exemple :

```
FOR i%=100 TO 300
  SETMOUSE i%,i%
  PAUSE 2
  PLOT MOUSEX,MOUSEY
  SHOWM
NEXT i%
```

--> déplace le curseur de la souris en diagonale sur l'écran en fixant des points.

HIDEM
SHOWM

Les instructions HIDEM et SHOWM permettent d'activer et de désactiver le curseur de la souris. Le fait d'appeler une boîte d'alerte ou une autre routine AES active automatiquement la souris.

Pour les sorties sur écran (PRINT) et les instructions graphiques (VDI, LINE-A), ce curseur de la souris est désactivé.

HIDEM permet de bloquer la réactivation automatique du curseur de la souris à la suite de déplacements (entre les instructions).

Exemple :

```
REPEAT
  IF MOUSEK=1
    SHOWM
  ENDIF
  IF MOUSEK=2
    HIDEM
  ENDIF
UNTIL MOUSEK=3
```

--> Le curseur de la souris est affiché si vous appuyez sur le bouton gauche de la souris. Il disparaît au contraire de l'écran si vous appuyez sur le bouton droit. Le fait d'appuyer simultanément sur les deux boutons met fin au programme.

STICK m
STICK(p)
STRIG(p)

```
m,p : iexp
```

L'ATARI ST dispose de deux interfaces (ports) pour la connexion de la souris et du joystick (manette de jeux). Sur le port 0 peuvent être lues les coordonnées de la souris ou du joystick, sur le port 1 uniquement des données de joystick.

L'instruction STICK0 commute sur un mode dans lequel le port 0 renvoie des coordonnées de la souris. Avec STICK1, les ports 0 et 1 renvoient des coordonnées de joystick. L'emploi de l'instruction STICK est généralement inutile car un STICK0 est automatiquement exécuté lorsque des coordonnées de souris sont réclamées ou bien un STICK1 lorsque des coordonnées de joystick sont réclamées. Dans les programmes testant le joystick, il est recommandé d'effectuer un STICK0 avant tout appel des fonctions AES (sélecteurs d'alerte, sélecteurs d'objet).

La fonction STICK(p) sert justement à tester la position du joystick. Avec p = 0, c'est la position du joystick sur le port 0, avec p = 1 c'est la position du joystick sur le port 1 qui sera renvoyée. STICK renvoie dans ce cas une valeur entre 0 et 10 qui indique la direction du mouvement du joystick d'après les règles suivantes :

```
5 1 9
  \|/
4--0--8
  /|\
6 2 10
```

STRIG(p) détermine l'état du bouton Feu (TRUE = actionné, sinon FALSE).

Exemple :

```
STICK 1              !Tester les coordonnées du joystick
'                     si le joystick est connecté sur le port 1
REPEAT
  sens%=STICK(1)
  feu!=STRIG(1)
  SELECT sens%
  CASE 4
    PRINT "Gauche"
  CASE 8
    PRINT "Droite"
  CASE 2
    PRINT "Bas"
  CASE 1
    PRINT "Haut"
  ENDSELECT
UNTIL feu!
WHILE STRIG(1)
WEND                 !Attend que le bouton feu soit relâché
```

--> Les messages du joystick sont désactivés par commutation sur la souris, c'est-à-dire que l'ordinateur ne remarque pas que le ou les bouton(s) feu ne sont pas actionnés. Pour y remédier, on peut, comme dans notre exemple, attendre à la fin du programme que les deux boutons feu aient été relâchés ou bien au contraire faire attendre le programme au début jusqu'à ce que l'état des boutons feu se modifie. Exemple ci-dessous.

```
PRINT "Veuillez actionner le bouton Feu"
WHILE STRIG(0)
WEND
REPEAT
UNTIL STRIG(0)
WHILE STRIG(0)
WEND
```

--> Complète l'exemple précédent.

IMPRIMANTE

LPRINT expression
LPOS(x)
HARDCOPY

```
(expression : aexp ou sexp ou une combinaison des deux)
(x : avar, argument fictif)
```

LPRINT expression permet de sortir des données sur l'imprimante. LPRINT fonctionne exactement comme l'instruction PRINT. Toutes les variantes de PRINT peuvent également être employées ici, si ce n'est qu'il n'est pas possible de positionner la tête d'écriture avec PRINT AT(x,y) mais LPRINT USING peut être employée.

LPOS(X) renvoie le nombre de caractères sortis depuis le dernier CR envoyé à l'imprimante (cf. POS(x)).

HARDCOPY fait sortir le contenu de l'écran sur une imprimante appropriée. C'est la routine de copie d'écran du système d'exploitation qui est ici utilisée. Cette routine peut également être appelée en appuyant simultanément sur les touches <ALTERNATE> et <HELP>. Il existe des programmes de drivers pour des imprimantes à 9 aiguilles non compatibles avec EPSON qui ne réagissent pas à l'instruction HARDCOPY (XBIOS(20)). La solution est dans ce cas SDPOKE &H4EE,0.

Exemples :

```
LPRINT
LPRINT "Test"
PRINT LPOS(x)
```

--> envoie un saut de ligne sur l'imprimante (s'il y en a une de connectée et sous tension), écrit le mot "Test" sur l'imprimante puis affiche sur l'écran la position actuelle de la tête d'impression.

```
FOR i%=20 TO 180 STEP 10
  CIRCLE 320,200,i%
NEXT i%
HARDCOPY
```

--> dessine des cercles concentriques sur l'écran et sort cette image sur l'imprimante.

GENERATION DE SONS

SOUND can,vol,note,octave,delai
SOUND can,vol,#per,delai
WAVE voix,env,forme,duree,delai

```
(can,vol,note,octave,delai,per,voix,env,forme,duree : iexp)
```

Les instructions SOUND et WAVE permettent de gérer les 3 générateurs de son de l'ATARI ST. SOUND attend à cet effet comme premier paramètre l'expression "can" qui fixe le canal voulu (1 à 3). Le volume (1 à 15) est fixé par le second paramètre "vol". Les paramètres "note" et "octave" servent à fixer la note voulue (1 à 12) de l'octave voulue (1 à 8). La table suivante indique la signification des codes de notes :

Note	Do	Do#	Ré	Ré#	Mi	Fa	Fa#	Sol	Sol#	La	La#	Si
Code	1	2	3	4	5	6	7	8	9	10	11	12

Une variante de l'instruction SOUND permet d'indiquer une période de vibration à l'aide de l'expression numérique "per" qui doit cependant être précédée du caractère #. "per" peut revêtir une valeur de 0 à 4095. Cette valeur se calcule à l'aide de la formule suivante lorsque la fréquence en Hertz est connue :

```
per=ROUND(125000/frequence)
```

Le *la* du diapason (440 Hz) peut ainsi être produit de deux façons différentes :

```
SOUND 1,15,10,4,250
SOUND 1,15,#284,250
SOUND 1,0,0,0,0
```

Les deux variantes de l'instruction admettent un dernier paramètre "delai" qui fixe en 1/50 de seconde le temps devant s'écouler avant l'exécution de la prochaine instruction (5 secondes dans notre exemple). La génération de sons peut être terminée avec l'instruction SOUND 1,0,0,0,0 (comme dans notre exemple) ou avec n'importe quelle autre instruction SOUND telle par exemple que celle produite lorsqu'une touche est frappée. Ce clic de la touche peut être désactivé avec :

```
SPOKE &H484, BCLR(PEEK(&H484),0)
```

Il peut être activé avec :

```
SPOKE &H484, BSET(PEEK(&H484),0)
```

WAVE permet de générer des sons sur plusieurs voix. "voix" est la somme des valeurs de bits correspondant aux 3 voix et des valeurs de bit fixant la fréquence du générateur de bruit :

```
 1 = voix 1
 2 = voix 2
 4 = voix 3
 8 = bruit pour la première voix
16 = bruit pour la seconde voix
32 = bruit pour la troisième voix
```

A cette somme peut être additionnée la période du générateur de bruit (0 à 31) multipliée par 256. "env" indique bit par bit, suivant le même principe que pour "voix", quelle voix doit être modulée par une courbe d'enveloppe. La forme de la courbe d'enveloppe peut être définie par "forme" d'après les règles suivantes :

```
0 à 3 = descente linéaire
4 à 7 = montée linéaire, interrompue
    8 = descente en dent de scie
    9 = descente linéaire (comme 0 à 3)
   10 = triangulaire, chute au début
   11 = descente linéaire, sautes de volume
   12 = montée en dent de scie
   13 = montée linéaire avec tenue
   14 = triangulaire, montée au début
   15 = montée linéaire, interrompue (comme 4 à 7)
```

La période de la courbe d'enveloppe est fixée par "durée". Comme pour l'instruction SOUND, "délai" fixe le temps devant s'écouler avant exécution de la prochaine instruction en 1/15 de seconde. Pour SOUND comme pour WAVE, les paramètres placés en fin d'instruction qui ne doivent pas être modifiés peuvent être omis.

Exemple :

```
SOUND 1,15,1,4,20
SOUND 2,15,4,4,20
SOUND 3,15,8,4,20
WAVE 7,7,0,65535,300
```

--> Une voix est produite sur chaque canal de son et modulée à l'aide de WAVE.

7. COMMANDES DU PROGRAMME

Le présent chapitre décrit toutes les instructions de commande d'un programme. Il commence par les instructions de décision qui permettent un traitement conditionnel des instructions (IF, THEN, ELSE, ENDIF, ELSEIF). Ces instructions servent donc uniquement à tester si une condition logique déterminée est vérifiée ou non.

Les instructions de décisions multiples (SELECT, CASE, DEFAULT, ENDSELECT, CONT) s'apparentent à ce premier groupe d'instructions mais il s'agit là d'instructions pour lesquelles le critère de décision ne peut pas seulement être vrai ou faux mais peut aussi recevoir n'importe quelles valeurs entraînant des réactions diversifiées.

La section suivante vous présentera les instructions de boucle du GFA-BASIC, qui permettent de répéter l'exécution de sections du programme. GFA-BASIC 3.0 dispose d'un très grand nombre de types de boucles (FOR TO STEP NEXT, REPEAT UNTIL, DO LOOP ou ENDLOOP, DO WHILE, DO UNTIL, LOOP WHILE, LOOP UNTIL, EXIT IF).

Pour une programmation structurée, la constitution de sous-programmes joue un rôle éminent. Les instructions PROCEDURE, GOSUB (@) et RETURN ou ENDSUB permettent de créer de nouvelles instructions sous forme de sous-programmes. Il est également possible de définir des fonctions. Les instructions DEFFN et FN permettent en effet de définir des fonctions présentant le caractère de formules. Les instructions FUNCTION, ENDFUNC et RESULT offrent des possibilités encore plus souples puisqu'elles permettent de constituer de véritables sous-programmes renvoyant une valeur de réponse.

Cette même section décrira également la gestion des variables dans les sous-programmes. Il s'agit tout d'abord de la possibilité de définir des variables locales (LOCAL) qui ne seront identifiées que dans une partie seulement du programme. Les variables peuvent être transmises à des sous-programmes sous forme de valeurs (call by value). L'instruction VAR permet cependant également de transmettre les variables elles-mêmes (call by reference) de façon à ce que celles-ci puissent être modifiées par le sous-programme sans qu'il soit nécessaire d'employer leur nom de variable global.

La section consacrée aux branchements conditionnés par des événements traite de deux événements caractéristiques du GFA-BASIC. Le premier est le fait d'appuyer simultanément sur les touches *Control, Alternate* et sur la touche *Shift* de gauche, ce qui entraîne normalement un arrêt du programme. Le second événement est l'apparition d'une erreur dans l'exécution du programme. C'est à ce propos que seront décrites les possibilités de programmation d'une routine de traitement des erreurs.

GFA-BASIC 3.0 permet d'appeler un sous-programme après un délai choisi. C'est à cela que servent les instructions EVERY et AFTER.

La fin du chapitre vous présente le branchement inconditionnel GOTO, les instructions de délai PAUSE et DELAY ainsi que différentes possibilités d'arrêt du programme (QUIT, SYSTEM, END, EDIT, NEW, STOP). La dernière section traite des instructions de surveillance du déroulement du programme (TRON, TROFF, TRON proc, TRACE$ DUMP, ERR$, ERROR).

INSTRUCTIONS DE DECISION

IF cond [THEN]
ELSE
ENDIF

(cond : bexp)

Ces instructions permettent de ne laisser agir une ou plusieurs instructions que lorsqu'une condition logique est remplie. L'exemple suivant illustre ce principe :

```
IF a=1 THEN
  PRINT "a égale 1"
  b=2
ENDIF
```

Dans ce cas, a=1 est la condition logique. Les instructions des lignes entre IF et ELSE ne seront exécutées que si cette condition logique est vérifiée. Si cette condition n'est pas remplie, le déroulement du programme se poursuivra après le mot d'instruction ENDIF. L'instruction THEN n'est pas indispensable. La formulation suivante sera donc également suffisante :

```
IF a=1
```
au lieu de
```
IF a=1 THEN
```

La construction suivante est un peu plus complexe :

```
IF a=1
  PRINT "a égale 1"
ELSE
  PRINT " a n'est pas égal à 1,"
  PRINT " mais à ";a
ENDIF
```

Dans ce cas, les instructions entre IF et ELSE seront exécutées si la condition logique après IF est remplie, après quoi le programme se poursuivra à la suite de l'instruction ENDIF.

Si cependant la condition après IF n'est pas remplie, les instructions entre ELSE et ENDIF seront exécutées. L'exécution du programme se poursuivra également après ENDIF. Toute expression différente de 0 est considérée comme logiquement vraie.

Exemple :

```
x=1
IF x
  PRINT "x est vrai"
ENDIF
INPUT y
IF x=9 OR ODD(y)
  PRINT "y est un nombre impair"
ELSE
  PRINT "y est un nombre pair"
ENDIF
```

--> Tout d'abord apparaît le texte 'x est vrai'. Un nombre est ensuite réclamé à l'utilisateur. Comme x ne peut pas être égal à 9 (puisqu'il vaut 1), le texte *'y est un nombre impair'* apparaît si vous avez entré un nombre impair, sinon c'est le texte *'y est un nombre pair'* qui est affiché.

ELSEIF cond

```
(cond : bexp)
```

L'instruction ELSEIF permet de représenter des instructions IF imbriquées de façon plus claire. L'exemple suivant réagit aux touches actionnées. Si la touche frappée est S, C ou E, des instructions remplaçant des sous-programmes de sauvegarde, de chargement ou d'entrée sont exécutées. Dans tous les autres cas, le texte *'Instruction inconnue'* apparaît. La version imbriquée de cet exemple se présente ainsi :

```
DO
  t$=CHR$(INP(2))
  '
  IF t$="C"
    PRINT "Chargement"
  ELSE
    IF t$="S"
      PRINT "Sauv
    ELSE
```

```
        IF t$="E"
          PRINT "Entrée"
        ELSE
          PRINT "Instruction inconnue"
        ENDIF
      ENDIF
    ENDIF
    '
  LOOP
```

L'emploi de ELSEIF permet d'obtenir un listing plus court en évitant d'avoir un trop grand nombre de retraits difficiles à identifier :

```
DO
  t$=CHR$(INP(2))
  '
  IF t$="C"
    PRINT "Chargement"
  ELSEIF t$="S"
    PRINT "Sauvegarde"
  ELSEIF t$="E"
    PRINT "Entrée"
  ELSE
    PRINT "Instruction inconnue"
  ENDIF
  '
LOOP
```

Ces structures de programme sont traitées d'après le principe suivant :

Si la condition après IF (ici t$="C") est remplie, les instructions entre IF et le prochain ELSEIF seront traitées (ici PRINT "Chargement") puis on sautera à l'instruction après ENDIF.

Si l'une des conditions après l'une des instructions ELSEIF est remplie, toutes les instructions entre ce ELSEIF et le prochain ELSE, ELSEIF ou ENDIF (s'il n'y a pas de ELSE) seront exécutées, après quoi on sautera à l'instruction à la suite de ENDIF.

Si ni la condition après IF ni l'une des conditions après ELSEIF ne sont vérifiées, ce sont les instructions entre ELSE et ENDIF qui seront exécutées (si ELSE il y a).

Exemples :

voyez plus haut.

BRANCHEMENTS MULTIPLES

ON x GOSUB proc1,proc2, ...

```
x : iexp
proc1, proc2 : noms de procédures sans paramètres
```

Cette instruction saute à la procédure numéro x dans la liste placée à la suite de GOSUB. x est ici une expression numérique dont les (éventuels) chiffres après la virgule seront ignorés. Si x est inférieur à un ou supérieur au nombre de noms de procédures à la suite de GOSUB, aucun sous-programme ne sera appelé. Après appel du sous-programme, le programme se poursuit à la suite de ON x GOSUB. Cette instruction ne permet pas de transmettre de paramètres aux procédures.

Exemple :

```
x=3
ON x GOSUB proc1,proc2,proc3
x=1
ON x+1 GOSUB proc1,proc2,proc3,proc4
```

--> La procédure *proc3* sera tout d'abord appelée, puis la procédure *proc2*.

SELECT x
CASE y [TO z] ou CASE y [,z,...]
CASE TO y
CASE y TO
DEFAULT
ENDSELECT
CONT

```
(x,y,z : iexp ou constante de chaine d'une longueur maximale de 4 caractères)
```

L'instruction SELECT permet des branchements en fonction de la valeur de l'expression numérique x. L'exemple suivant illustre la structure de programme qui en résulte :

```
x=0
SELECT x+2
CASE 1
  PRINT "x égale 1"
CASE 2 TO 4
```

```
    PRINT "x égale 2,3 ou 4"
  CASE 5,6
    PRINT "x égale 5 ou 6"
  DEFAULT
    PRINT "x est différent de 1,2,3,4,5 ou 6"
  ENDSELECT
```

--> Le message *'x égale 2, 3 ou 4'* est sorti sur le moniteur.

L'expression numérique après SELECT, qui représente la condition de branchement, est tout d'abord calculée (elle vaut ici 2). Les instructions CASE sont ensuite parcourues de haut en bas et on examine si la valeur actuelle de la condition de branchement figure parmi elles. Dans notre exemple, seul le 1 figure après la première instruction CASE. Comme la condition de branchement vaut 2, le programme saute donc au prochain CASE.

A la suite du second CASE figure '2 TO 4'. Cette condition sera donc remplie si le critère de branchement après SELECT revêt des valeurs entre 2 et 4 (limites comprises). Dans le cas présent, les instructions entre les second et troisième CASE seront donc exécutées. L'exécution du programme se poursuit ensuite après l'instruction ENDSELECT.

Le troisième CASE présente une autre variante pour l'indication des valeurs possibles du critère SELECT. Les valeurs voulues y sont en effet énumérées dans une liste, séparées par des virgules.

Si la teneur actuelle du critère de branchement n'est citée à la suite d'aucun CASE, ce sont les instructions entre DEFAULT et ENDSELECT qui seront traitées, à condition toutefois qu'une instruction DEFAULT ait été prévue. Vous pouvez aussi écrire OTHERWISE au lieu de DEFAULT, auquel cas l'interpréteur remplacera automatiquement ce mot par DEFAULT.

Après CASE peuvent cependant être spécifiées non seulement des constantes numériques mais aussi des constantes de chaînes d'une longueur maximale de quatre caractères. Si un seul caractère est spécifié, c'est sa valeur ASCII qui sera employée comme critère de branchement. Si deux caractères sont spécifiés, la valeur de ce critère se calculera comme suit :

valeur ASCII du premier caractère + 255 * valeur ASCII du second caractère.
On procède de même si trois ou quatre caractères sont spécifiés.

Exemple :

```
sortir!=FALSE
REPEAT
  frapp%=inp(2)
  SELECT frapp%
```

```
    CASE "a" TO "z"
      PRINT "La lettre minuscule "+chr$(frapp%)+" a été entrée"
    CASE "A" TO "Z"
      PRINT "La lettre majuscule "+chr$(frapp%)+" a été entrée"
    CASE 27
      sortir!=TRUE   !Fin du programme lorsque <Esc> actionnée
    DEFAULT
      PRINT "Une touche non autorisée a été actionnée !"
    ENDSELECT
  UNTIL sortir!
```

--> Le clavier est testé dans une boucle et un branchement est effectué en fonction de la touche actionnée. Le caractère correspondant est affiché ou bien l'entrée est annoncée comme incorrecte. Le fait d'appuyer sur la touche <Esc> sert de critère d'interruption pour la boucle.

L'exemple suivant illustre le rôle de l'instruction CONT qui n'a d'effet que lorsqu'elle figure devant une instruction CASE ou DEFAULT. Cette instruction CONT ne doit pas être confondue avec son homonyme qui sert à relancer un programme dont l'exécution a été interrompue temporairement.

Exemple :

```
x=1
SELECT x
CASE 1
  PRINT "x égale 1"
  CONT
CASE 2
  PRINT "x égale 2"
CASE 1,3
  PRINT "x égale 3"
DEFAULT
  PRINT "x est différent de 1,2 et 3"
ENDSELECT
```

--> *'x égale 1'* et *'x égale 2'* apparaissent sur le moniteur.

L'instruction CONT a pour effet de faire ignorer l'instruction CASE ou DEFAULT placée après elle. Dans cet exemple, la valeur après la première instruction CASE est bien égale à la condition de branchement (puisque nous avons défini en première ligne x=1). C'est donc l'instruction PRINT "x égale 1" qui est exécutée. Ensuite vient cependant une instruction CONT qui a pour effet de faire ignorer la ligne CASE 2. Par conséquent, bien que la condition de branchement ne soit pas remplie puisque x n'est pas égal à 2, l'instruction PRINT "x égale 2" sera exécutée. Lorsque la prochaine instruction CASE ou DEFAULT est rencontrée, le programme saute

alors normalement après ENDSELECT, et ce même si (comme c'est le cas ici) la condition de branchement après le CASE suivant correspond à la teneur actuelle du critère de branchement.

Exemple :

```
SELECT INP(2)
CASE "a" TO "g"
  PRINT "a à g"
DEFAULT
  PRINT "default"
ENDSELECT
```

--> Si vous appuyez sur l'une des touches a, b, c, d, e, f ou g, le texte *'a à g'* apparaît, pour toute autre touche (sauf les touches de commutation du clavier), c'est *'default'* qui apparaît.

CASE permet de combiner les différentes possibilités, de sorte qu'il n'est pas nécessaire d'écrire :

```
SELECT a$
CASE "a" TO "z"
  CONT
CASE "A" TO "Z"
  CONT
CASE "à","â","ç","é","è","ê","ë","î","ï","ô","ù","û"
  PRINT "OK"
ENDSELECT
```

car les lignes suivantes suffiront :

```
CASE "a" TO "z", "A" TO "Z", "à","â","ç","é","è","ê","ë","î","ï", "ô","ù","û", ...
...
...
```

BOUCLES

FOR, TO, STEP, NEXT
REPEAT, UNTIL
WHILE, WEND
DO, LOOP, DO WHILE, DO UNTIL, LOOP WHILE, LOOP UNTIL
EXIT IF

GFA-BASIC 3.0 dispose d'une gamme de types de boucles exceptionnellement étendue.

On distingue normalement des boucles *a priori* et des boucles *a posteriori*. Les boucles a priori sont des boucles dans lesquelles le critère de sortie de la boucle est testé avant même l'entrée dans la boucle. Avec les boucles a posteriori, au contraire, ce critère n'est testé qu'à la fin du corps même de la boucle. La conséquence pratique de cette différence est que les boucles a posteriori sont toujours exécutées au moins une fois.

GFA-BASIC dispose tout d'abord de deux types de boucles a posteriori usuels, les boucles FOR-NEXT et REPEAT-UNTIL. Un troisième type de boucle standard est représenté par la boucle a priori WHILE-WEND. Un quatrième type de boucle est représenté par DO-LOOP. Il s'agit là en fait d'une boucle sans fin, sans critère de sortie, mais cette instruction peut être employée avec de nombreuses variantes en GFA-BASIC. En effet, aussi bien DO que LOOP peuvent être suivies des extensions WHILE et UNTIL de sorte qu'il devient également possible de construire des boucles testant une condition logique aussi bien au début qu'à la fin de la boucle.

Par ailleurs, il est possible, pour tous les types de boucles indiqués ci-dessus d'implanter dans le corps même de la boucle autant de critères de sortie que vous le souhaitez (EXIT IF).

FOR i=a **TO** e [**STEP** s]
  (Instructions)
NEXT i
DOWNTO

```
(i : avar)
(a,e,s : aexp)
```

La boucle FOR-NEXT (également appelée boucle de comptage) sert à répéter l'exécution d'un groupe d'instructions (le corps de la boucle) placé entre les mots d'instructions FOR et NEXT. A cet effet, on affecte tout d'abord la valeur de départ a à la variable de comptage i. Les instructions du corps de la boucle sont ensuite traitées jusqu'à ce que NEXT soit atteint. La variable i est alors augmenté de la valeur *s* qui figure après STEP. Si aucun STEP n'est spécifié, i est augmentée de la valeur un. On examine alors si i dépasse la valeur e. Si c'est le cas, on passe à l'instruction à la suite de NEXT, sinon on recommence le traitement de la boucle à partir de la première instruction du corps de la boucle. Cette opération se répète jusqu'à ce que i soit supérieur ou égal à e.

La conséquence de ce principe de fonctionnement est que i sera toujours égal, après sortie de la boucle, à la première valeur ayant dépassée le critère de sortie, à moins que i ait été dès le départ fixé sur une valeur dépassant ce critère. Le corps de la boucle FOR est toujours traité au moins une fois.

Au lieu du pas STEP s avec s=-1, vous pouvez aussi employer l'instruction DOWNTO au lieu de TO mais il n'est pas possible d'employer STEP avec DOWNTO.

Il est conseillé d'employer des variables entières comme variable de comptage i car les instructions de boucle sont dans ce cas traitées plus rapidement qu'avec des variables à virgule flottante. Il va de soi qu'il n'est cependant pas possible d'employer des variables entières comme variable de comptage si le pas n'est pas une valeur entière.

Au lieu du mot d'instruction NEXT suivi de la variable de comptage (i% par exemple), vous pouvez aussi écrire *endfor i%*, ce que l'interpréteur remplacera par NEXT i%.

Exemples :

```
FOR i=1 TO 10
  PRINT i'
NEXT i
FOR i=-1 DOWNTO -10
  PRINT i'
NEXT i
```

--> Ecrit les nombres 1 2 3 4 5 6 7 8 9 10 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 sur l'écran.

```
a$="M*o*t* *d*e* *t*e*s*t"
FOR j=1 TO LEN(a$) STEP 2
  PRINT MID$(a$,j,1);
NEXT j
```

--> Mot de test apparaît sur l'écran.

REPEAT
(Instructions)
UNTIL cond

```
(cond : bexp)
```

REPEAT - UNTIL *cond* encadrent un groupe d'instructions qui seront exécutées jusqu'à ce le critère logique *cond* donne 'vrai'.

Lorsque l'instruction REPEAT est atteinte dans le programme, les instructions placées à la suite sont traitées jusqu'à ce que UNTIL soit atteint. On teste alors si la condition logique après UNTIL est vérifiée.

Si c'est le cas, les instructions à la suite de UNTIL sont exécutées. Si par contre *cond* donne 'faux' (0), l'exécution du programme reprend à la suite de l'instruction REPEAT.

Les instructions entre REPEAT et UNTIL sont traitées au moins une fois, à moins que cette boucle a posteriori ne soit abandonnée prématurément avec EXIT IF ou GOTO.

Au lieu de UNTIL *cond*, vous pouvez aussi écrire ENDREPEAT *cond*, ce que l'interpréteur remplacera automatiquement par UNTIL.

Exemples :

```
REPEAT
UNTIL MOUSEK
```

--> Attend qu'un bouton de la souris soit actionné.

```
i=1
REPEAT
  INC i
  j=SQR(i)
UNTIL i>10 and FRAC(j)=0
PRINT i
```

--> Le nombre 16 est sorti sur l'écran.

WHILE cond
(Instructions)
WEND

```
(cond : bexp)
```

Les instructions WHILE et WEND encadrent un groupe d'instructions qui seront traitées aussi longtemps que la condition logique *cond* sera remplie. Lorsque GFA-BASIC rencontre l'instruction WHILE, il examine la condition logique placée à la suite. Si cette condition est vérifiée, les instructions entre WHILE et WEND sont traitées. Lorsque WEND est atteint, le programme saute à nouveau à WHILE et le cycle reprend jusqu'à ce que *cond* soit fausse. Au lieu de l'instruction WEND, vous pouvez aussi écrire ENDWHILE, ce que l'interpréteur remplace automatiquement par WEND.

Exemple :

```
WHILE INKEY$=""
  PLOT MOUSEX,MOUSEY
WEND
```

--> Dessine avec la souris jusqu'à ce que vous frappiez une touche. Si un caractère figure déjà dans le buffer clavier, aucun point n'est fixé.

DO
(Instructions)
LOOP

Les instructions DO LOOP produisent une boucle sans fin. Le programme traite les instructions entre DO et LOOP puis retourne à l'instruction après DO dès qu'il rencontre LOOP.

La boucle ne peut être abandonnée par le programme qu'à l'aide des instructions EXIT IF, GOTO ou d'instructions de fin du programme. Au lieu de l'instruction LOOP, vous pouvez aussi écrire ENDDO, ce que l'interpréteur remplace automatiquement par LOOP.

Exemple :

```
DEFFILL 1,2,4
DO
  MOUSE mx,my,mk
  IF mk
    PBOX mx,my,mx+25,my+25
  ENDIF
LOOP
```

--> Dessine des rectangles pleins sur la position actuelle de la souris lorsque vous appuyez sur un bouton de la souris.

DO WHILE cond
DO UNTIL cond
LOOP WHILE cond
LOOP UNTIL cond

```
(cond : bexp)
```

Les instructions DO ... LOOP peuvent être complétées à l'aide des suffixes UNTIL et WHILE.

En tête de la boucle, DO WHILE aura pour effet de ne faire exécuter l'intérieur de la boucle que tant que *cond* sera vraie. Si la boucle commence au contraire par DO UNTIL, on n'y entrera que si la condition *cond* n'est pas remplie.

LOOP WHILE fait revenir le programme à DO tant que *cond* est vraie. Avec LOOP UNTIL au contraire, *cond* doit être fausse pour qu'il y ait retour au début de la boucle.

Les conditions sont donc posées a priori avec DO ou a posteriori avec LOOP.

```
DO WHILE cond                          WHILE cond
.                                      .
.                  équivaut à          .
.                                      .
LOOP                                   WEND

DO                                     REPEAT
.                                      .
.                                      .
.                                      .
LOOP UNTIL cond                        UNTIL cond
```

Les variantes de l'instruction DO, DO WHILE et DO UNTIL peuvent être combinées à volonté avec LOOP, LOOP WHILE et LOOP UNTIL, de sorte que ces instructions permettent de construire neuf types de boucles différents.

Exemples :

```
DO
LOOP UNTIL MOUSEK
```

--> Attend qu'un bouton de la souris soit actionné.

```
DO UNTIL MOUSEK=2
  DO WHILE MOUSEK=1
    LINE 0,0,MOUSEX,MOUSEY
  LOOP
LOOP UNTIL INKEY$="a"
```

--> Dessine des lignes lorsque le bouton gauche de la souris est tenu enfoncé. Le programme prend fin lorsque le bouton droit de la souris ou la touche 'a' sont actionnés.

```
DO UNTIL EOF(#1)
  INPUT #1,a$
LOOP
```

--> Lit des chaînes de caractères de façon séquentielle sur le canal 1 jusqu'à ce que la fin du fichier soit atteinte.

```
WHILE NOT EOF(#1)
  INPUT #1,a$
WEND
```

--> La construction avec WHILE WEND est plus lente parce qu'une opération NOT est nécessaire en plus.

EXIT IF cond

```
cond : bexp
```

EXIT IF permet de sauter en dehors d'une boucle si la condition booléenne *cond* est remplie. Le type de boucle peut être librement choisi.

Contrairement aux versions plus anciennes, EXITIF est admis même à l'intérieur de IF-ENDIF et de SELECT-ENDSELECT.

Exemple :

```
DO
  EXIT IF MOUSEK
LOOP
REPEAT
  EXIT IF INKEY$="x"
UNTIL FALSE
```

--> Met fin au programme après qu'on ait tout d'abord actionné un bouton de la souris puis la touche 'x'.

PROCEDURES ET FONCTIONS

Sous GFA-BASIC 3.0, les sous-programmes reçoivent des noms comme dans n'importe quel autre langage de programmation moderne. Ces sous-programmes peuvent aussi recevoir des paramètres et ils peuvent aussi être appliqués à différentes variables lors de différents appels à l'aide de paramètres VAR.

Il est possible de transmettre aussi bien la valeur d'une variable que la variable elle-même (VAR). Dans le second cas, la variable transmise peut alors être modifiée directement par la procédure. La première possibilité est dite *"call by value"*, la seconde *"call by reference"*.

L'emploi de variables locales est également possible, ce qui permet d'éviter tout risque de confusion de nom entre des variables utilisées dans les procédures ou fonctions d'appel et les variables apparaissant uniquement à l'intérieur des procédures ou fonctions appelées.

DEFFN permet de définir sur une ligne des fonctions qui seront par la suite appelées sous un nom donné à l'aide de FN. Il est également possible de définir des fonctions sur plusieurs lignes (FUNCTION) qui ne représentent en fait qu'une forme particulière de procédures renvoyant un résultat (à travers RETURN).

GOSUB proc [(par1,par2,...)]
PROCEDURE proc [(var1,var2,...)]
RETURN

```
proc : nom de la procédure
par1,par2 : sexp,aexp
var1,var2 : svar,avar
```

Entre les mots d'instruction PROCEDURE et RETURN figurent les instructions d'un sous-programme. - la suite de PROCEDURE figure le nom d'un sous-programme ou éventuellement la liste des variables à recevoir. Pour appeler une procédure, il suffit d'indiquer son nom au début d'une ligne, en le faisant éventuellement suivre de paramètres appropriés, placés entre parenthèses. Pour souligner qu'il ne s'agit pas d'instructions GFA-BASIC, un "@" facultatif ou le mot-clé "GOSUB" peut être placé devant le nom de la procédure. Cela est cependant obligatoire si le nom de la procédure peut être confondu avec une instruction GFA-BASIC (on écrira par exemple @stop, @rem).

Les paramètres transmis peuvent être des constantes, des variables ou des expressions. Pour les variables, ce sont aussi bien leur valeur que ces variables elles-mêmes qui peuvent être transmises (voyez VAR).

Lorsque l'instruction RETURN est atteinte lors de l'exécution du programme, le programme saute à l'instruction après GOSUB. Au lieu du mot d'instruction PROCEDURE, vous pouvez aussi écrire SUB ou encore ENDPROC ou ENDSUB au lieu de RETURN. L'interpréteur rétablira automatiquement les mots d'instructions standard.

Exemple :

```
GOSUB slow_print("** Manuel de **")
@slow_print("* GFA-BASIC 3.0 *")
@slow_print("GFA-SYSTEMTECHNIK")
'
PROCEDURE slow_pr    (t$)
```

```
    LOCAL i%
    FOR i%=1 TO LEN(t$)
      PRINT MID$(t$,i%,1);
      PAUSE 3
    NEXT i%
    PRINT
  RETURN
```

--> Affichage lent, caractère par caractère, d'une chaîne de caractères.

```
a=8
@racine_cubique(a)
PRINT a
'
PROCEDURE racine_cubique(VAR x)
  x=x^(1/3)
RETURN
```

--> Calcule la racine cubique de 8 et affiche 2 sur l'écran.

LOCAL var1 [,var2,var3,...]

```
(var1,var2,var3 : avar,svar)
```

L'instruction LOCAL permet de restreindre le domaine de validité des variables. Les variables énumérées à la suite de LOCAL ne seront valides que dans la procédure dans laquelle figure cette instruction LOCAL ainsi que dans tous les sous-programmes appelés par cette procédure.

Après LOCAL peuvent aussi être désignées des variables qui valent dans le programme principal, c'est-à-dire des variables globales. Ces variables globales ne pourront dans ce cas être appelées à l'intérieur du sous-programme mais elle redeviendront disponibles, sans avoir été modifiées, dès que la procédure sera abandonnée.

Les variables transmises à une procédure ou une fonction sont toujours locales.

Exemple :

```
x=2
GOSUB test
PRINT x,y
'
PROCEDURE test
  LOCAL x,y
```

```
    x=3
    y=4
  RETURN
```

--> Sur l'écran apparaissent les nombres 2 et 0.

@fonc [(par1,par2,...)]
FUNCTION fonc [(var1,var2,...)]
RETURN exp
ENDFUNC

```
fonc : nom de la fonction
par1,par2 : sexp,aexp
var1,var2 : svar,avar
xp : sexp, aexp
```

Entre les mots d'instructions FUNCTION et ENDFUNC figurent les instructions d'un sous-programme (de même qu'avec PROCEDURE). FUNCTION est suivi du nom du sous-programme ainsi, éventuellement que de la liste des variables à transmettre. Le sous-programme est appelé à l'aide de l'arobas @ ou de FN suivi du nom de la fonction ainsi, éventuellement, que d'une liste de paramètres.

Les paramètres transmis peuvent être des constantes, des variables et des expressions. Pour les variables, ce sont aussi bien leur valeur que ces variables elles-mêmes qui peuvent être transmises (voyez VAR).

Lorsque l'instruction RETURN est atteinte lors de l'exécution du programme, la valeur calculée par la fonction vient remplacer l'appel de la fonction.

RETURN peut aussi apparaître plusieurs fois dans une fonction, par exemple si vous employez IF, etc..., mais il n'est pas permis de terminer une fonction (avec ENDFUNC) sans RETURN.

Un signe $ à la fin du nom de fonction désigne les fonctions dont le résultat est une chaîne de caractères.

Exemple :

```
fl#=@loop_fac(15)
fr#=@recur_fac(10)
'
PRINT "Boucle :    fac(15) = ";fl#
PRINT "Récursion : fac(10) = ";fr#
'
FUNCTION loop_fac(f%)
```

```
  w=1
  FOR j%=1 TO f%
    MUL w,j%
  NEXT j%
  RETURN w
ENDFUNC
'
FUNCTION recur_fac(f%)
  IF f%<2
    RETURN 1
  ELSE
    RETURN f%*@recur_fac(PRED(f%))
  ENDIF
ENDFUNC
```

--> Calcul des factorielles de 10 et de 15 dans une boucle, puis avec récursion.

DEFFN fonc [(x1,x2,...)] = ext
FN fonc [(y1,y2,...)]

```
fonc,x1,x2 : var
ext,y1,y2 : exp
```

L'instruction DEFFN sert à définir des fonctions sur une ligne. La définition de la fonction figure dans l'expression *ext* qui peut avoir pour résultat aussi bien une expression numérique qu'une expression de chaîne. Une fois définie, la fonction est appelée avec FN fonc ou avec l'arobas @.

Les variables figurant dans la définition de la fonction sont transmises comme paramètres. A cet effet, les noms de variables apparaissant dans la définition de la fonction sont notés x1,x2,... Lors de l'appel de la fonction, les paramètres y1,y2,... peuvent être transmis sous forme d'expressions numériques ou d'expressions de chaînes.

Si les variables x1,x2,... sont également définies comme globales, elles ne peuvent être appelées à l'intérieur de l'expression *ext* car elles recevront dans le cadre de *ext* les valeurs transmises comme paramètres. Les variables x1,x2,... sont donc toujours des variables locales pour la fonction.

L'instruction FN peut également être remplacée par @. Les fonctions peuvent être imbriquées à loisir mais avec DEFFN elles ne peuvent l'être de façon récursive (car aucune condition d'interruption n'est possible).

Exemples :

```
DEFFN test(y,a$)=x-y+LEN(a$)
x=2
PRINT @test(4,"abcdef")
```

--> Le nombre 4 (2-4+6) apparaît sur l'écran.

```
DEFFN first_last$(a$)=LEFT$(a$)+RIGHT$(a$)
b$=@first_last$("TEST")
PRINT b$
```

--> Le texte "TT" apparaît sur le moniteur.

```
DEFFN puissance_quatre(x)=x^4
DEFFN racine_quatre(x)=x^(1/4)
PRINT @racine_quatre(@puissance_quatre(1024))
```

--> Le nombre 1024 apparaît.

BRANCHEMENTS EN FONCTION DE CERTAINS EVENEMENTS

Cette section traitera de deux types d'événements qui peuvent être testés en GFA-BASIC. Les événements qui ne sont pas caractéristiques du GFA-BASIC, tels que le test du clic de la souris, la sélection dans des menus déroulants, etc..., seront traités dans d'autres chapitres (programmation des menus et des fenêtres, bibliothèques AES).

Le premier des deux événements évoqués ici sera le fait d'appuyer simultanément sur les touches *Control*, *Shift* et *Alternate*. Le second type d'événement sera l'apparition d'une erreur.

ON BREAK
ON BREAK CONT
ON BREAK GOSUB proc

```
(proc : nom d'une procédure)
```

Ces trois instructions définissent la réaction au fait d'appuyer simultanément sur les touches *Control*, *Shift* (touche *Shift* de gauche uniquement) et *Alternate*.

Le fait d'appuyer sur cette combinaison de touches entraîne normalement l'arrêt du programme mais il peut aussi servir à sauter à une procédure déterminée. A cet effet, la procédure *proc* à appeler doit être définie avec ON BREAK GOSUB.

ON BREAK CONT entraîne une absence de réaction au fait d'appuyer sur cette combinaison de touches. ON BREAK réactive l'état normal (arrêt du programme).

Exemple :

```
ON BREAK GOSUB test
PRINT "Veuillez appuyer sur CONTROL, SHIFT (gauche) et ALTERNATE"
DO
LOOP
'
PROCEDURE test
  PRINT "Et voilà"
  ON BREAK
RETURN
```

--> Un message invite à appuyer sur la combinaison de touches. Lorsque celle-ci est effectivement actionnée, la procédure réactive la routine *Break* habituelle.

ON ERROR
ON ERROR GOSUB proc
RESUME [NEXT]
RESUME [mar]

```
(proc : nom d'une procédure)
(mar : nom d'une marque)
```

Lorsqu'une erreur apparaît (que ce soit une erreur TOS ou une erreur spécifique du GFA-BASIC), cela entraîne normalement l'affichage d'un message d'erreur et l'arrêt du programme. ON ERROR GOSUB permet de déclencher un saut à la procédure *proc* lorsque survient une erreur. Il est ainsi possible de définir dans cette procédure la réaction à adopter en cas d'erreur.

L'instruction ON ERROR rétablit le traitement habituel des erreurs, c'est-à-dire l'affichage d'un message d'erreur et l'arrêt du programme. Lorsqu'une erreur se produit, une instruction ON ERROR est automatiquement exécutée. Pour qu'il soit possible de réagir à plusieurs erreurs successives, il est donc nécessaire d'employer encore une fois l'instruction ON ERROR GOSUB *proc* dans la routine de traitement des erreurs.

L'instruction RESUME permet une réaction d'un type particulier aux erreurs apparues.

Elle n'a de sens qu'à l'intérieur de la procédure de traitement des erreurs. RESUME NEXT permet de sauter à la prochaine instruction après celle ayant entraîné l'erreur. RESUME *mar* entraîne un saut à la marque *mar*. L'instruction RESUME sans indication de NEXT ni d'une marque entraîne un saut à l'instruction ayant provoqué l'erreur. Si une erreur fatale (voyez FATAL) était apparue, seule l'instruction RESUME *mar* peut être utilisée mais pas RESUME NEXT ni RESUME sans indication de marque.

Exemple :

```
ON ERROR GOSUB traitement_des_erreurs
ERROR 5
PRINT "et encore une fois ..."
ERROR 5
PRINT "n'est jamais atteint"
'
PROCEDURE traitement_des_erreurs
  PRINT "Ok, erreur interceptée."
  RESUME NEXT
RETURN
```

--> Les textes *'Ok, erreur interceptée.'* et *'et encore une fois'* apparaissent sur l'écran, puis le message d'erreur *'Racine carrée uniquement pour les nombres positifs'* provoqué par ERROR 5. La marque pour RESUME peut figurer aussi bien dans une procédure que dans le programme principal.

ERROR x
ERR
ERR$(x)
FATAL

```
(x : aexp)
```

ERROR permet de déclencher l'erreur numéro x (voyez la table des messages d'erreur en annexe). Cela sert par exemple à tester une routine de traitement des erreurs.

La variable système ERR contient le numéro de l'erreur apparue. Elle permet à l'intérieur d'une routine de traitement des erreurs de définir des réactions spécifiques pour certaines erreurs.

La fonction ERR$ renvoie la chaîne de caractères du message d'erreur correspondant à l'erreur numéro x.

La variable système FATAL est vraie si l'erreur apparue s'est produite à une adresse non identifiée par l'interpréteur. Cela peut survenir par exemple lorsque l'erreur s'est produite lors de l'exécution d'une routine du système d'exploitation. La conséquence pour le programme est qu'une instruction RESUME ou RESUME NEXT ne peut plus être traitée correctement à la suite d'une erreur fatale.

Exemples :

```
ON ERROR GOSUB traitement_des_erreurs
INPUT "Quelle erreur souhaitez-vous : ",e
ERROR e
'
PROCEDURE traitement_des_erreurs
  PRINT "C'était l'erreur numéro : ";ERR
  IF FATAL
    PRINT "qui est une erreur fatale."
  ENDIF
RETURN
```

--> Réclame un numéro d'erreur à l'utilisateur et affiche ce numéro.

```
~FORM_ALERT(1,ERR$(100))
```

--> Affiche sur l'écran le message d'erreur correspondant au code 100 (c'est-à-dire le message de Copyright) sous forme d'une boîte d'alerte.

PROGRAMMATION DES INTERRUPTIONS

EVERY ticks **GOSUB** proc
EVERY STOP
EVERY CONT
AFTER ticks **GOSUB** proc
AFTER STOP
AFTER CONT

```
(ticks : iexp)
(proc : nom d'une procédure)
```

Les instructions EVERY et AFTER permettent de faire appeler des procédures après écoulement d'un certain délai *ticks*. L'instruction EVERY entraîne un appel de la procédure *proc* toutes les *ticks* unités de temps ; AFTER entraîne un appel unique de cette procédure une fois *ticks* unités de temps écoulées.

Le délai est ici exprimé en deux-centièmes de seconde (ticks = 200 correspondra donc à un délai d'une seconde). Le branchement à la procédure indiquée ne peut toutefois être effectué que toutes les quatre unités de temps, de sorte que la précision horaire effective est seulement d'un cinquantième de seconde.

EVERY STOP désactive l'appel de la procédure lorsque le délai est écoulé, EVERY CONT le rétablit au contraire. Les instructions AFTER STOP et AFTER CONT jouent le même rôle pour AFTER. Sur un plan interne, GFA-BASIC exécute ces instructions à travers le vecteur *etv_timer* ($400).

Ce n'est qu'une fois achevée l'exécution de chaque instruction qu'il est testé si l'une de ces procédures doit être appelée. Les instructions lentes telles que INP(2), QSORT, les opérations de fichier et autres du même genre peuvent donc paralyser ces routines.

Exemples :

```
EVERY 4 GOSUB lines
lignes!=TRUE
GRAPHMODE 3
DEFFILL 1,0
PLOT MOUSEX,MOUSEY
REPEAT
  IF MOUSEK=1
    EVERY STOP
  ELSE
    EVERY  CONT
  ENDIF
  DRAW TO MOUSEX,MOUSEY
UNTIL MOUSEK=2
'
PROCEDURE lines
  INC y%
  LINE 320,y%,639,y%
  IF y%=399
    y%=0
  ENDIF
RETURN
```

--> Fait courir des lignes de haut en bas dans la moitié supérieure de l'écran et permet simultanément de dessiner avec la souris. Le fait d'appuyer sur le bouton gauche de la souris active ou désactive les lignes baladeuses. Le programme peut être interrompu en appuyant sur le bouton droit de la souris.

```
PRINT "Dans 3 secondes apparaitra un texte"
PRINT "si vous ne frappez aucune touche."
AFTER 600 GOSUB texte
```

```
REPEAT
UNTIL INKEY$<>"" OR dehors !
AFTER STOP
'
PROCEDURE texte
  PRINT
  PRINT "Voici le texte"
  dehors!=TRUE
RETURN
```

--> Si aucune touche n'est actionnée dans les trois secondes suivant le lancement du programme, un texte apparaît. Si une touche est actionnée, le programme s'arrête.

DIVERS

(REM, GOTO, PAUSE, DELAY)
(END, EDIT)
(STOP)
(SYSTEM, QUIT)

REM x
' x
<Ligne d'instruction> ! x

```
(x : texte quelconque)
```

Sur une ligne commençant par REM ou ' peut figurer n'importe quel texte. Ces textes ne sont pas soumis au contrôle de syntaxe opéré par l'éditeur et ils ne sont pas pris en compte lors de l'exécution du programme. D'autre part, à la fin d'une ligne d'instruction, un commentaire quelconque peut être ajouté à la suite du point d'exclamation '!'. Ce symbole ne peut toutefois être utilisé à cet effet dans les lignes d'instructions DATA. INLINE ne permet pas non plus l'insertion de commentaires.

Exemple :

```
REM Commentaire
' PRINT "Commentaire"
PRINT "REM"  ! Commentaire
```

--> Sur l'écran apparaît le mot REM.

GOTO mar
mar:

```
(mar : marque définie par le programmeur)
```

A l'aide d'une marque *mar:* vous pouvez définir des emplacements du programme auxquels vous pourrez ensuite sauter à l'aide de l'instruction GOTO. L'exécution du programme se poursuivra alors dans la position de la marque. La marque peut se composer de lettres, chiffres, caractères souligné et de points. Elle peut aussi, contrairement aux noms de variables, commencer par un chiffre mais elle doit en tout cas se terminer par un double point lors de sa définition. Ce double point n'est toutefois pas indiqué lorsque la marque est appelée.

Il n'est pas possible de sauter à l'aide de l'instruction GOTO à l'intérieur de procédures ou de fonctions ou bien de sauter hors de ces procédures ou fonctions. Cette possibilité s'applique également aux boucles FOR-NEXT. Les instructions GOTO sont un vestige des anciennes versions du BASIC qui ne disposaient pas de possibilités de programmation structurée. Elles rendent les programmes très vite illisibles et il est donc conseillé d'éviter d'y recourir.

Exemple :

```
PRINT "Emplacement 1"
GOTO marque_de_saut
PRINT "Emplacement 2"
marque_de_saut:
PRINT "Emplacement 3"
```

--> Sur l'écran apparaissent les textes *"Emplacement 1"* et *"Emplacement 3"*.

PAUSE x
DELAY x

```
x : aexp
```

L'instruction PAUSE suspend l'exécution du programme pendant un délai de x / 50 secondes. DELAY a le même effet si ce n'est que l'exécution du programme est suspendue pendant x secondes (précision théorique en millisecondes). DELAY emploie la routine GEM EVNT_TIMER et est donc recommandée pour les programmes GEM.

Exemple :

```
PRINT "Début"
PAUSE 100
PRINT "Une pause"
DELAY 2
PRINT "Fin"
```

--> Le texte *'Début'* apparaît, puis deux secondes plus tard *'Une pause'* et encore 2 secondes après *'Fin'*.

END
EDIT
STOP

Ces instructions servent à terminer un programme. L'instruction END met fin à l'exécution du programme et fait apparaître un sélecteur d'alerte avec le texte *'Fin du programme'*. Une fois la seule possibilité de choix offerte sélectionnée par l'utilisateur, GFA-BASIC 3.0 revient à l'éditeur.

EDIT met fin à l'exécution du programme et revient immédiatement à l'éditeur.

STOP fait apparaître un sélecteur avec les choix ARRET et CONT. Après sélection de CONT, l'exécution du programme reprend. Après sélection de ARRET par l'utilisateur, GFA-BASIC revient en mode direct où les valeurs des variables peuvent être modifiées ou testées, après quoi l'exécution du programme peut être reprise en sélectionnant CONT.

Exemple :

```
x=3
STOP
PRINT x
```

--> Sélectionnez le bouton 'ARRET' lorsqu'apparaît la boîte d'alerte correspondante. Entrez alors les instructions suivantes en mode direct :

```
PRINT x
```

--> Le nombre 3 apparaît. Entrez :

```
x=4
CONT
```

--> La dernière instruction du listing (PRINT x) est maintenant exécutée. C'est le nombre 4, c'est-à-dire la valeur spécifiée en mode direct, qui apparaît.

NEW

Cette instruction efface un programme. En mode direct, une demande de confirmation intervient auparavant par mesure de sécurité. Sous l'éditeur, cette instruction peut être déclenchée avec *Shift-F4* ou par un clic de la souris (avec demande de confirmation).

LOAD f$

```
f$ : sexp
```

L'instruction LOAD sert à charger un programme GFA-BASIC. L'expression de chaîne f$ contient le chemin d'accès du fichier voulu. A défaut de spécification de cette extension, c'est .GFA qui est prédéfinie.

Exemple :

```
LOAD "A:\TEST.GFA"
```

--> Charge le fichier de programme TEST.GFA à partir du répertoire racine du lecteur A.

SAVE f$
PSAVE f$

```
f$ : sexp
```

L'instruction SAVE stocke un fichier de programme sous le nom spécifié dans f$. Si l'instruction PSAVE est employée, le fichier spécifié est stocké avec une protection *anti-List* (c'est-à-dire que lorsque le fichier sera rechargé avec LOAD il ne pourra plus être listé et sera immédiatement exécuté). A défaut de spécification d'une extension, c'est .GFA qui sera employée.

Exemple :

```
SAVE "A:\TEST.GFA"
```

--> Sauvegarde le fichier de programme actuel sous le nom "TEST.GFA" sur le lecteur A.

LIST [f$]
LLIST [f$]

```
f$ : sexp
```

L'instruction LIST affiche le programme actuel sur l'écran. En option, un chemin d'accès peut être spécifié sous lequel le fichier de programme sera alors sauvegardé en format ASCII. Les fichiers de programme qui devront être insérés ultérieurement dans d'autres programmes avec MERGE doivent être stockés en format ASCII, avec LIST ou SAVE,A (dans la barre des menus).

A défaut de spécification d'une extension, c'est .LST qui est prédéfinie. LLIST permet de sortir le programme actuel sur l'imprimante.

L'impression du listing peut être interrompue en arrêtant l'imprimante. Un délai d'environ 30 secondes s'écoule alors avant que reprenne l'exécution du programme ou bien avant le retour sous l'éditeur (voyez aussi LLIST et les instructions de point dans la section consacrée à l'éditeur). Il est cependant également possible de détourner la sortie (cf. OPEN).

Exemple :

```
LIST "A:\TEST.LST"
```

--> Sauvegarde le programme actuel sous le nom "TEST.LST" en format ASCII sur le lecteur A.

```
.ll 70
.pl 66
LLIST
```

--> Sort le programme actuel sur l'imprimante avec une longueur de ligne de 70 caractères et 66 lignes par page.

CHAIN f$

```
f$ : sexp
```

L'instruction CHAIN permet de charger un programme GFA-BASIC dans la mémoire de travail et lance le programme. A défaut de spécification d'une extension, c'est .GFA qui est prédéfinie.

Exemple :

```
CHAIN "A:\COMPL.GFA"
```

--> Charge le fichier de programme COMPL.GFA et lance le programme.

RUN [f$]

```
f$ : sexp
```

L'instruction RUN lance le programme actuel. Si un nom de fichier complet est spécifié, le programme voulu sera d'abord chargé puis lancé.

Exemple :

```
RUN "A:\PARTIE_2.GFA"
```

--> Charge et lance à partir du lecteur A le programme appelé PARTIE_2.GFA.

SYSTEM [n]
QUIT [n]

```
n : iexp
```

Les instructions SYSTEM et QUIT ont le même effet, à savoir qu'elles mettent fin au déroulement du programme et quittent GFA-BASIC. Contrairement à ce qui se passait sous les versions 2.xx, SYSTEM et QUIT permettent, en option, de renvoyer une valeur entière sur 2 octets. Ce mot vaut zéro si l'interpréteur a été abandonné dans des conditions normales et il est renvoyé au programme d'appel (habituellement le bureau).

Par convention, zéro signale une exécution sans erreur, un nombre positif de 16 bits désigne une erreur interne ou un avertissement et un nombre négatif sur 16 bits correspond généralement au message d'erreur correspondant du système d'exploitation. Cette convention n'est cependant pas respectée par tous les programmes.

Exemple :

```
RESERVE 100
PRINT EXEC(0,"GFABASIC.PRG","","")
```

--> Lancez ce petit programme (Shift-F10), après quoi un GFA-BASIC sera à nouveau chargé.

Dans ce GFA-BASIC, entrez l'instruction :

```
QUIT 23
```

--> et lancez-le. Affichage : 23 et fin du programme pour le premier GFA-BASIC chargé.

TRAITEMENT DES ERREURS

TRON
TRON #n
TROFF

```
(n : iexp)
```

L'instruction TRON (trace on) sert à faire afficher sur l'écran chaque instruction exécutée. Cette liste peut cependant également être écrite dans un fichier si vous spécifiez un numéro de canal. Elle peut aussi être envoyée sur l'imprimante ou sur l'interface sérielle, etc..., si vous spécifiez un numéro de canal. L'instruction TROFF met fin à cette situation.

Exemples :

```
PRINT "Début :"
TRON
FOR i%=1 TO 5
  PRINT i%
NEXT i%
TROFF
PRINT "Fin."
```

--> Le mot *'Début :'* apparaît, puis les nombres 1 à 5 et les instructions servant à les afficher, enfin le mot *'Fin'*.

```
OPEN "O",#1,"\tron.lst"
TRON #1
FOR i%=1 TO 10
  PRINT i%
NEXT i%
TROFF
CLOSE #1
'
OPEN "O",#2,"prn:"
TRON #2
```

```
FOR i%=10 TO 630 STEP 10
  LINE i%,0,i%,100
NEXT i%
TROFF
CLOSE #2
```

--> Les nombres 1 à 10 apparaissent, ainsi que les instructions nécessaires à cet effet et qu'une série de lignes verticales espacées de 10 points écran. Cette sortie d'instructions se fait sur disquette ou sur l'imprimante.

TRON proc
TRACE$

```
proc : nom d'une procédure
```

L'instruction TRON *proc* permet de spécifier une procédure qui devra être appelée avant exécution de chaque instruction. La variable TRACE$ contient la prochaine instruction devant être exécutée. En liaison avec TRACE$, TRON *proc* permet un dépistage des erreurs extrêmement efficace. C'est ainsi que certaines variables peuvent être sorties sur l'imprimante en fonction de la prochaine instruction devant être exécutée, ce qui permet de suivre précisément les modifications d'une variable au cours du déroulement du programme.

Il importe de veiller à ce que la procédure TRON apporte le moins de perturbations possible. Eviter donc les instructions PRINT dans des masques d'écran (TEXT, ATEXT, ...) ou l'emploi de routines VDI (à cause de l'emploi du GDOS ou de DEFTEXT-LINE-A), etc...

Exemple :

```
TRON proc_tr
GRAPHMODE 3
DO UNTIL MOUSEK
  x1%=100+RAND(200)
  y1%=100+RAND(100)
  x2%=200+RAND(200)
  y2%=200+RAND(100)
  PBOX x1%,y1%,x2%,y2%
LOOP
'
PROCEDURE proc_tr
  IF BIOS(11,-1) AND 4    !Touche Control
    adr%=XBIOS(2)
    BMOVE adr%+1280,adr%,4*1280
    PRINT AT(1,5);SPACE$(80);
```

```
      PRINT AT(1,5);LEFT$(TRACE$,79);
      PAUSE 20
    ENDIF
  RETURN
```

--> Ce programme dessine sur l'écran des rectangles répartis au hasard. Le fait d'appuyer sur la touche *Control* fait apparaître les instructions traitées sur l'écran. Le fait d'appuyer sur un bouton de la souris met fin au programme.

DUMP [a$ [TO b$]]

```
a$,b$ : sexp
```

L'instruction DUMP permet de faire afficher le contenu de variables au cours du déroulement du programme ou bien de faire sortir des listes des labels, procédures et fonctions. A cet effet, l'expression de chaîne a$ peut revêtir les valeurs suivantes :

Exemples :

```
DUMP
```

--> Sort les valeurs de toutes les variables et le dimensionnement de tous les tableaux.

```
DUMP "a"
```

--> Comme ci-dessus mais uniquement pour les variables ou tableaux commençant par 'a'.

```
DUMP ":"
```

--> Sort la liste de tous les labels (marques) avec indication pour chacun du numéro de la ligne sous l'éditeur où la marque est définie. La variante (:b) ne sortira que la liste des marques commençant par 'b'.

```
DUMP "@"
```

--> Sort la liste de toutes les procédures et fonctions, chaque nom étant suivi du numéro de la ligne où chaque procédure ou fonction figure sous l'éditeur :

```
nom_proc @ 100        (procédure)
nom_fonc FN 200       (fonction avec valeur de réponse numérique)
nom_fonc $ FN 300     (fonction avec chaine comme valeur de réponse)
```

Les labels, procédures et fonctions qui ne sont plus définis sont indiqués sans numéro de ligne. Si le listing du programme est sauvegardé avec l'option SAVE,A puis rechargé, ces noms indéfinis disparaissent. Toutefois les noms de labels, procédures et fonctions qui sont utilisés mais ne sont pas définis apparaissent sans numéros de lignes.

Sous l'éditeur, vous pouvez sauter aux numéros de lignes indiqués à la suite des noms avec *Control+G*.

Les chaînes sont affichées avec un maximum de 60 caractères. Si une chaîne de caractères dépasse cette limite, elle est suivie du caractère '>', sinon d'un "". Les caractères de commande (c'est-à-dire les caractères dont le code ASCII est inférieur à 32) sont représentés par un '.'

Les sorties indiquées ci-dessus peuvent également être redirigées vers un fichier. Dans ce cas, un nom de fichier doit être spécifié dans b$.

A défaut d'extension, c'est .DMP qui sera supposée.

8. GRAPHISME

L'Atari ST offre trois différents modes graphiques, un mode noir et blanc et deux modes couleur. Les coordonnées des instructions graphiques et les couleurs qui peuvent être affichées avec ces instructions dépendent de la résolution actuelle. C'est pourquoi nous vous proposons maintenant une récapitulation des modes graphiques:

	Points écran (Coordonnées)	**Couleurs (Registres de couleurs)**
Résolution basse :	320 x 200 (0 à 319) (0 à 199)	16 (0 à 15) sur 512 couleurs
Résolution moyenne :	640 x 200 (0 à 639) (0 à 199)	4 (0 à 3) sur 16 couleurs
Haute résolution :	640 x 400 (0 à 639) (0 à 399)	2 (0 et 1)

La première section présente les instructions pour la sélection des couleurs (SETCOLOR, COLOR). Viennent ensuite les instructions de définition pour la sélection et la création de formes de curseur de la souris, de marquages, de motifs de remplissage, de cadres et de types de lignes les plus divers (DEFMOUSE, DEFMARK, DEFFILL, BOUNDARY, DEFLINE).

La section suivante présente les instructions CLIP pour limiter les sorties graphiques ainsi que les instructions graphiques générales pour le dessin de différentes formes géométriques de base (PLOT, LINE, BOX, CIRCLE, ELLIPSE). Elle décrit également les possibilités de dessin de polygones (POLYLINE, POLYMARK, POLYFILL) et de texte dans le graphisme (TEXT). La section se termine enfin par la présentation de l'instruction FILL.

La dernière section de ce chapitre traite de la manipulation de sections de l'écran avec SGET, SPUT, GET et PUT.

INSTRUCTIONS DE DEFINITION

SETCOLOR registre,rouge,vert,bleu
SETCOLOR registre,proportions
COLOR couleur

```
(registre,rouge,vert,bleu,proportions,couleur : iexp)
```

La première variante de SETCOLOR fixe la proportion des couleurs de base rouge, vert et bleu dans un registre de couleur déterminé. A cet effet, l'intensité des composantes de la couleur est fixée à l'aide d'une échelle allant de 0 (pour faible) à 7 (pour forte). Le nombre de couleurs disponibles dépend de la résolution actuelle. Avec la seconde variante, le réglage des couleurs est calculé d'après la formule suivante : proportions = rouge * 256 + vert * 16 + bleu, les valeurs de rouge, vert et bleu étant à nouveau tirées de l'échelle de 0 à 7.

La définition des proportions des couleurs dans les registres de couleur ne présente naturellement d'intérêt que sous les résolutions couleur. Pour la résolution monochrome, lorsqu'une autre valeur que 0 ou 1 est indiquée pour proportions, les nombres pairs ont le même effet que 0 et les nombres impairs le même effet que 1.

L'instruction COLOR fixe la couleur d'écriture, l'expression numérique couleur contenant une valeur de 0 à 15 en fonction de la résolution actuelle.

Exemple :

```
SETCOLOR 0,0
```

--> Sur le moniteur monochrome apparaît une écriture blanche sur fond noir.

DEFMOUSE symbole
DEFMOUSE motif_bits$

```
(symbole : iexp)
(motif_bits$ : sexp)
```

La première variante de l'instruction sélectionne le symbole de la souris parmi 8 formes prédéfinies :

```
0 --> flèche
1 --> double parenthèse
2 --> abeille
```

```
3 --> main avec index pointé
4 --> main ouverte
5 --> réticule mince
6 --> réticule épais
7 --> réticule encadré
```

La seconde variante sert à définir un curseur de souris personnel à partir d'un point d'action, d'une couleur de masque, d'une couleur de curseur et de deux motifs de bits pour la forme du masque et la forme du curseur de la souris, spécifiés par l'utilisateur dans une chaîne de caractères. Le point d'action est le point du curseur de la souris dont les coordonnées sont gérées comme correspondant à la position de la souris. Lorsque la position de la souris est réclamée, ce sont les coordonnées du point d'action qui sont renvoyées.

Il convient de noter à cet égard que toutes ces valeurs doivent être transmises sous forme de mots. L'instruction MKI$ peut être employée à cet effet, de sorte que motif_bits$ peut être ainsi constitué :

```
motif_bits$=MKI$(coordonnée x du point d'action)
           +MKI$(coordonnée y du point d'action)
           +MKI$(1)      ! normal, -1=XOR
           +MKI$(couleur_du_masque)
           +MKI$(couleur_du_curseur)
           +masque$  (motif bits du masque)
           +curseur$ (motif bits du curseur)
```

masque$* et *curseur$ se composent chacun de 16 mots dont chacun définit le motif de bits d'une ligne.

Exemple :

```
DEFMOUSE 2
REPEAT
UNTIL MOUSEK
m$=MKI$(0)+MKI$(0)+MKI$(1)+MKI$(0)+MKI$(1)
FOR i%=1 TO 16
  m$=m$+MKI$(65535)
NEXT i%
FOR i%=1 TO 16
  m$=m$+MKI$(1)
NEXT i%
PBOX 0,0,200,100
DEFMOUSE m$
REPEAT
UNTIL MOUSEK=2
```

--> Une abeille apparaît tout d'abord comme curseur de la souris. Après que le bouton gauche de la souris ait été actionné, le curseur de la souris a la forme d'un trait. Sur le fond noir apparaît en haut à gauche la forme d'un rectangle, ce qui est dû à la définition du masque. La boucle d'attente est abandonnée en appuyant sur le bouton droit de la souris.

DEFMARK [couleur], [type], [taille]

```
(couleur,type,taille : iexp)
```

DEFMARK définit la couleur, le type et la taille des coins fixés avec l'instruction POLYMARK. Pour l'expression numérique couleur, des valeurs entre 0 et 15 sont admises en fonction de la résolution (voyez l'introduction de ce chapitre). *type* désigne les marquages suivants :

```
1 --> point
2 --> signe plus
3 --> étoile
4 --> rectangle
5 --> croix
6 --> losange
```

Les valeurs supérieures à 6 sélectionnent l'étoile comme type de marquage. Les valeurs pour la taille du marquage s'expriment en points écran. Si vous ne voulez utiliser que le second ou troisième paramètre de l'instruction, le premier paramètre peut être omis pourvu que les virgules de séparation ne soient pas oubliées. DEFMARK „4 signifiera par exemple que les deux premiers paramètres garderont leur teneur actuelle mais que la taille des marquages sera fixée sur 4.

Exemple :

```
DIM x%(1),y%(1)
x%(0)=50
y%(0)=50
x%(1)=150
y%(1)=150
DEFMARK 1,4,2
POLYMARK 2,x%(),y%()
DEFMARK ,3,4
POLYMARK 2,x%(),y%() OFFSET 30,0
```

--> Dessine deux couples de points avec des marquages finaux différents.

```
DEFFILL 1,f$
FILL 120,120
```

--> Dessine un rectangle avec le motif de remplissage prédéfini puis un autre avec un motif de remplissage aléatoire.

```
DO
  FOR j%=0 TO 15
    f$=""
    s%=BCHG(s%,j%)
    FOR i%=1 TO 16
      f$=f$+MKI$(s%)
    NEXT i%
    DEFFILL 1,f$
    PBOX 0,0,639,399
  NEXT j%
LOOP
```

--> Définit un motif de remplissage et sort un rectangle avec ce motif.

```
FOR j%=1 TO 64
  READ a%
  a$=a$+MKI$(a%)
NEXT i%
DEFFILL ,a$
PBOX 20,20,300,200
DATA -1,-1,-1,-1,-1,-1,-1,-1, 0, 0, 0, 0, 0, 0, 0, 0
DATA -1,-1,-1,-1, 0, 0, 0, 0,-1,-1,-1,-1, 0, 0, 0, 0
DATA -1,-1, 0, 0,-1,-1, 0, 0,-1,-1, 0, 0,-1,-1, 0, 0
DATA -1, 0,-1, 0,-1, 0,-1, 0,-1, 0,-1, 0,-1, 0,-1, 0
```

--> Crée un rectangle avec motif de remplissage sous forme de lignes verticales. Suivant la résolution que vous utilisez, la taille de ces lignes sera différentes.

BOUNDARY n

```
(n : iexp)
```

L'instruction BOUNDARY emploie la fonction VDI *vsf_perimeter* et active ou désactive l'encadrement automatique de la surface de remplissage. Si n vaut zéro, l'encadrement est désactivé, si n est différent de zéro l'encadrement est activé.

DEFFILL [couleur] , [style] , [motif]
DEFFILL [couleur] , motif_bits$

```
(couleur,style,motif : iexp)
(motif_bits$ : sexp)
```

Cette instruction fixe le motif de remplissage pour les instructions PBOX, PCIRCLE, PELLIPSE, POLYFILL et FILL. Elle fixe la couleur, le style et le motif de remplissage, permettant ainsi de définir des motifs personnels. Pour la variable couleur, des valeurs entre 0 et 15 sont admises en fonction de la résolution (voyez l'introduction de ce chapitre). *style* correspond aux définitions suivantes :

```
0 --> vide
1 --> rempli
2 --> pointillé
3 --> hachuré
4 --> symbole ATARI
```

motif permet de sélectionner un des 24 motifs de point ou un des 12 motifs de ligne (voyez l'annexe : table des motifs de remplissage).

Avec la seconde variante de l'instruction sont transmis dans *motif_bits$*, les 32 octets d'informations de bits définissant un motif de remplissage de 16 points écran sur 16. Ces informations doivent se présenter en format de mot et il est donc conseillé d'avoir recours à l'instruction MKI$ pour les préparer.

Les paramètres placés devant peuvent être omis à condition de ne pas oublier néammoins les virgules de séparation. C'est ainsi que DEFFILL ,2,4 sélectionnera par exemple le motif de remplissage 2,4 tout en laissant la couleur de remplissage inchangée.

En mode couleur et en résolution moyenne, les 16 mots pour le premier plan de bits peuvent encore être suivis de 16 mots pour le second plan de bits. En résolution faible, ce seront même 4 * 16 = 64 mots pour les motifs de remplissage de plusieurs couleurs.

Exemples :

```
DEFFILL 1,2,4
BOX 50,50,100,100
FILL 70,70
FOR i=1 TO 16
  f$=f$+MKI$(RAND(65
NEXT i
BOX 100,100,150,150
```

Exemple :

```
DEFFILL 1,2,2
BOUNDARY 1          ! Activer encadrement
PBOX 50,50,100,100
BOUNDARY 0          ! Désactiver encadrement
PBOX 150,50,200,100
```

--> Dessine deux rectangles pleins avec et sans cadre.

DEFLINE [style], [epaisseur], [s_debut], [s_fin]

```
(style,epaisseur,s_debut,s_fin : iexp)
```

L'instruction DEFLINE définit l'apparence des lignes dessinées avec les instructions LINE, BOX, RBOX, CIRCLE, ELLIPSE et POLYLINE. Le premier paramètre *style* fixe le style de ligne. Vous avez ici le choix entre les motifs prédéfinis et les motifs définis par l'utilisateur. Les motifs prédéfinis suivants sont possibles :

```
0 --> Ligne dans la couleur du fond
1 --> Ligne continue
2 --> Ligne de tirets légèrement espacés
3 --> Ligne pointillée
4 --> Ligne points-tirets
5 --> Lignes de tirets très espacés
6 --> Motif de ligne : trait-point-point ...
```

Les motifs choisis par l'utilisateur peuvent être définis en indiquant une valeur de 16 bits dans laquelle chaque bit mis correspondra à un point fixé (en mode monochrome).

Le second paramètre *epaisseur* fixe l'épaisseur d'une ligne en points écran. L'épaisseur de la ligne ne peut être qu'une valeur impaire. Toute épaisseur de ligne paire sera interprétée comme le plus proche nombre impair inférieur.

s_debut et *s_fin* définissent les symboles de début et de fin du type de ligne. Vous disposez ici du choix suivant :

```
0 --> coin
1 --> flèche
2 --> arrondi
```

Les paramètres placés devant peuvent être omis à condition de ne pas oublier les virgules de séparation.

C'est ainsi que DEFLINE „1,1 définira par exemple la flèche comme symbole pour le début et la fin de la ligne tout en laissant *style* et *largeur* inchangés (voyez la table en annexe).

Exemple :

```
FOR i=1 TO 6
  DEFLINE i
  LINE 50,i*50,200,i*50
NEXT i
DEFLINE 1,1,1,2
FOR i=2 TO 12 STEP 2
  DEFLINE ,i
  LINE 250,i*25,400,i*25
NEXT i
DEFLINE -&X1010101010101010,1,0,0
LINE 500,10,500,390
VOID INP(2)
```

--> Des lignes seront dessinées avec les six styles prédéfinis. Ensuite seront produites des lignes de différentes épaisseurs. La ligne avec le motif défini par l'utilisateur sera pointillée. Le programme attend ensuite, pour terminer, qu'une touche soit actionnée.

DEFTEXT [couleur],[attr],[angle],[hauteur],[nofonte]

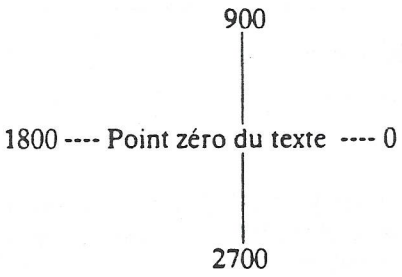
```
(couleur,attr,angle,hauteur,nofonte : iexp)
```

Cette instruction fixe l'apparence d'une chaîne de caractères devant être sortie avec TEXT.

Pour l'expression numérique couleur, des valeurs entre 0 et 15 sont admises en fonction de la résolution (voyez l'introduction de ce chapitre). *attr* sélectionne un ou plusieurs attributs de texte suivants, qui peuvent être combinés par addition des codes correspondants :

```
 0 --> normal        (normal)
 1 --> bold          (gras)
 2 --> light         (brillant)
 4 --> italic        (italique)
 8 --> underlined    (souligné)
16 --> outlined      (encadré)
```

angle fixe l'orientation de la sortie de texte. Sa valeur est exprimée en 1/10 de degrés dans le sens contraire des aiguilles d'une montre.



Les valeurs suivantes sont admises :

```
   0 --> Valeur prédéfinie, de gauche à droite
 900 --> de bas en haut
1800 --> renversé, de droite à gauche
2700 --> renversé, de haut en bas
```

hauteur fixe la hauteur d'une lettre du texte en points écran. Pour les jeux de caractères normaux, seules les hauteurs d'écriture suivantes sont cependant bien lisibles :

```
 4 --> hauteur d'écriture des icônes
 6 --> écriture fine
13 --> hauteur normale d'écriture
32 --> écriture très haute
```

nofonte permet enfin de spécifier le numéro d'un jeu de caractères voulu qui doit avoir été correctement installé au préalable (voyez aussi GDOS : VST_LOAD_FONT, VQT_NAME, ...).

Exemple :

```
FOR i|=0 TO 5
  DEFTEXT 1,2^i|,0,13
  TEXT 100,i|*16+100,"Voici l'attribut du texte "+STR$(i|)
NEXT i|
~INP(2)
```

--> Sort un texte d'exemple avec différents attributs.

GRAPHMODE n

```
n : iexp
```

L'instruction GRAPHMODE définit de quelle façon les sorties graphiques doivent être combinées entre elles sur l'écran. L'expression numérique n sélectionne ici l'un des 4 modes possibles :

```
1 --> replace              (remplacer)
2 --> transparent          (transparent)
3 --> xor                  (inverser)
4 --> reverse transparent  (inverser et transparent)
```

Si n vaut 1, ce qui correspond au mode prédéfini, le nouveau motif recouvrera toujours le précédent. Avec n égal à 2, les ancien et nouveau motifs seront combinés avec OR.

Avec n égal à 3 sera fixé tout point écran qui n'était pas fixé précédemment et sera au contraire annulé tout point écran qui était déjà fixé (les motifs sont donc inversés par une combinaison XOR).

En mode 4, le nouveau motif sera tout d'abord inversé puis sera combiné avec OR.

Exemple :

```
FOR i%=1 TO 4
  GRAPHMODE i%
  DEFFILL 1,3,8
  PBOX 150*i%-100,10,150*i%,100
  DEFFILL 1,2,10
  PBOX 150*i%-140,50,150*i%-40,150
NEXT i%
```

--> Cet exemple montre 4 paires de rectangles de motifs différents qui sont combinés entre eux sous les 4 modes.

INSTRUCTIONS GRAPHIQUES GENERALES

Cette section vous présentera tout d'abord les possibilités de limitation des sorties graphiques à l'aide des variantes de l'instruction CLIP.

Les instructions graphiques générales PLOT, LINE, BOX, RBOX, CIRCLE et ELLIPSE dessinent des points, des lignes, des rectangles, des rectangles à coins arrondis, des cercles et des ellipses.

PBOX, PRBOX, PCIRCLE et PELLIPSE permettent de remplir ces figures avec des couleurs ou des motifs.

POLYLINE dessine un polygone dont les coins peuvent se voir attribuer différents symboles à l'aide de POLYMARK. POLYFILL remplit ce polygone avec un motif, dans une couleur déterminée. POINT fournit la couleur d'un point déterminé de l'écran. FILL remplit une zone de l'écran délimitée. TEXT permet de sortir des chaînes de caractères dans n'importe quel emplacement de l'écran. CLS vide l'écran. Pour ces instructions graphiques, l'origine des coordonnées se situe dans le coin supérieur gauche de l'écran. Les coordonnées peuvent recevoir des valeurs sortant des limites de la résolution de l'écran mais seules les parties visibles du dessin seront naturellement affichées. La section se terminera par la présentation de l'instruction BITBLT.

CLIP x,y,w,h [**OFFSET** x0,y0]
CLIP x1,y1 **TO** x2,y2 [**OFFSET** x0,y0]
CLIP #n [**OFFSET** x0,y0]
CLIP OFFSET x,y
CLIP OFF

```
x,y,w,h,x0,y0,x1,y1,x2,y2,n : iexp
```

Ce groupe d'instructions sert à la délimitation appelée *'clipping'*. Il s'agit d'une technique permettant de limiter les sorties graphiques à une zone rectangulaire quelconque de l'écran. CLIP active le 'clipping' pour les instructions graphiques VDI, alors que ACLIP ne l'active que pour les routines graphiques LINE-A. Pour fixer cette zone de l'écran (clipping rectangle), doivent être spécifiées les coordonnées des coins opposés en diagonale ou bien la largeur et la hauteur du rectangle de délimitation.

Il existe différentes variantes de l'instruction CLIP. CLIP x,y,w,h permet de spécifier la coordonnée gauche x 'x' et la coordonnée supérieure y 'y' ainsi que la largeur 'w' et la hauteur du rectangle de délimitation.

Une autre possibilité est constituée par l'instruction CLIP x1,y1 TO x2,y2. Ce sont ici les coordonnées des coins opposés en diagonale (x1,y1) et (x2,y2) qui doivent être spécifiées.

La troisième variante permet de limiter les sorties aux limites de la fenêtre numéro 'n'.

Avec les variantes indiquées, le suffixe de l'instruction optionnelle OFFSET x0,y0 permet de fixer l'origine des sorties graphiques sur le point de coordonnées x0,y0. L'instruction OFFSET x0,y0 peut aussi être utilisée seule au même effet, l'origine des sorties graphiques étant donc également placé sur le point (x0,y0).

Le 'clipping' est désactivé à l'aide de l'instruction CLIP OFF. La limitation des sorties graphiques ne s'applique pas aux instructions GET, PUT, BITBLT ni aux appels LINE-A ou aux instructions AES.

PLOT x,y
LINE x1,y1,x2,y2
DRAW [**TO**] [x,y]
DRAW [x1,y1] [**TO** x2,y2] [**TO** x3,y3] [**TO** ...]

```
(x,y,x1,y1,x2,y2: iexp)
```

PLOT dessine un point de coordonnées x,y sur l'écran. LINE trace une ligne du couple de coordonnées x1,y1 au couple de coordonnées x2,y2. Le style et la couleur de cette ligne peuvent être définis avec DEFLINE et COLOR.

DRAW x,y équivaut à l'instruction PLOT. DRAW TO x,y trace une ligne du dernier point fixé aux coordonnées x,y. Peu importe à cet égard que ce dernier point ait été fixé auparavant avec PLOT, LINE ou DRAW. Une autre variante de l'instruction DRAW est représentée par DRAW x1,y1 TO x2,y2, qui équivaut à l'instruction LINE. D'autres coordonnées peuvent encore être indiquées qui permettent alors, par exemple, de produire des polygones. La dernière variante de l'instruction DRAW permet enfin de spécifier des instructions qui ne sont pas sans rappeler certaines instructions graphiques du 'LOGO' (Turtle graphics ou graphisme de tortue) ou du langage standard des plotters 'HPGL' de Hewlett-Packard. Il est ainsi possible de simuler un plotter sur l'écran.

Exemples :

```
x=50
y=50
couleur=POINT(x,2*y)
PRINT couleur
PLOT x,2*50
LINE 200,200,400,100
PRINT POINT(x,100)
~INP(2)
```

--> Dessine un point et une ligne. Détermine la couleur du point.

```
DO
  MOUSE mx,my,mk
  IF mk=1
    DRAW TO mx,my
  ENDIF
```

```
    EXIT IF mk=2
  LOOP
```

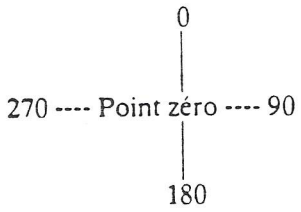
--> Lorsque le bouton gauche de la souris est actionné, une liaison est dessinée entre les coordonnées actuelles de la souris, mx,my, et le dernier point fixé. La boucle est abandonnée lorsque le bouton droit de la souris est actionné.

DRAW expression
DRAW(i)
SETDRAW

```
i : iexp
expression :   une séquence de sexp et aexp, dont le premier élément doit être
               sexp ; séparation avec virgule, point-virgule ou apostrophe.
```

DRAW déplace sur l'écran un crayon imaginaire et permet le dessin relatif. Cela ressemble beaucoup aux instructions graphiques du langage Logo. Les paramètres des différentes instructions sont toujours des variables à virgule flottante qui peuvent également être spécifiées dans des chaînes. Les instructions suivantes sont disponibles :

FD n	FORWARD	déplace le 'crayon' de n points 'en avant'
BK n	BACKWARD	déplace le 'crayon' de n points 'en arrière'
SX x	SCALE X	gradue le 'déplacement du crayon' avec FD et BK
SY y	SCALE Y	à raison du facteur spécifié La graduation avec SX et SY n'agit que sur les instructions FD et BK. SX0 ou SY0 désactivent cette graduation (plus rapide que la graduation avec le facteur 1 (SX1, SY1)).
LT a	LEFT TURN	spécifie l'angle 'a' de la rotation du dessin sur la gauche
RT a	RIGHT TURN	de même pour la droite
TT a	TURN TO	définit l'angle absolu 'a' suivant la définition suivante :



Les indications de a=angle se font en degrés.

MA x,y	MOVE ABSOLUTE	amène le 'crayon' sur les coordonnées absolues x et y.

DA x,y	DRAW ABSOLUTE	amène le 'crayon' sur les coordonnées absolues x et y et dessine une ligne dans la couleur actuelle de la dernière position au point (x,y).
MR xr,yr	MOVE RELATIVE	comme 'MA' mais par rapport à la dernière position.
DR xr,yr	DRAW RELATIVE	comme 'DR' mais par rapport à la dernière position.

L'instruction SETDRAW x,y,a représente une abréviation de l'expression DRAW "MA",x,y,"TT",a.

CO c	COLOR	fixe la couleur 'c' comme couleur de dessin (voyez aussi les paramètres pour l'instruction COLOR).
PU	PEN UP	relève le 'crayon'.
PD	PEN DOWN	abaisse le 'crayon'.

Vous disposez en outre des fonctions de test suivantes :

DRAW(0) fournit	la position x (sous forme de valeur à virgule flottante)
DRAW(1) fournit	la position y (sous forme de valeur à virgule flottante)
DRAW(2) fournit	l'angle en degrés (sous forme de valeur à virgule flottante)
DRAW(3) fournit	la graduation sur l'axe des x (sous forme de valeur à virgule flottante)
DRAW(4) fournit	la graduation sur l'axe des x (sous forme de valeur à virgule flottante)
DRAW(5) fournit	le flag du crayon (-1=PD, 0=PU)

Exemples :

```
DRAW "ma 160,200 tt0"     ! commencer en 160,200 avec l'angle 0
FOR i&=3 TO 10
  coin(i&,90)             ! dessiner un polygone de i angles
NEXT i&
'
PROCEDURE coin(n&,r&)     ! n=nombre d'angles, r=longueur de côté
  LOCAL i&
  FOR i&=1 TO n&
    DRAW "fd",r&,"rt",360/n&
  NEXT i&
RETURN
```

--> Dessine des polygones.

```
FOR i=0 TO 359 STEP 8
  SETDRAW 320,200,i
  GRAPHMODE 3
  DRAW "fd 45 rt 90 fd 45 rt 90 fd 45 rt 90 fd 45"
  DRAW "bk 90 rt 90 bk 90 rt 90 bk 90 rt 90 bk 90"
  GRAPHMODE 1
  DRAW "fd 45 rt 90 fd 45 rt 90 fd 45 rt 90 fd 45"
  DRAW "bk 90 rt 90 bk 90 rt 90 bk 90 rt 90 bk 90"
NEXT i
```

--> Dessine un petit et un grand rectangles tournant autour de leur propre axe.

```
l%=48
' losange 1
DRAW "ma60,100 tt45"
DRAW "fd",l%," rt90 fd",l%," rt90 fd",l%," rt90 fd",l%," rt90"
' rhomboïde, étroit
DRAW "mr100,0 tt45"
DRAW "sx0.5 sy0"
DRAW "fd",l%," rt90 fd",l%," rt90 fd",l%," rt90 fd",l%," rt90"
' losange, large
DRAW "mr100,0 tt45"
DRAW "sx0 sy0.5"
DRAW "fd",l%," rt90 fd",l%," rt90 fd",l%," rt90 fd",l%," rt90"
' losange, large et haut
DRAW "mr100,0 tt45"
DRAW "sx3 sy2"
DRAW "fd",l%," rt90 fd",l%," rt90 fd",l%," rt90 fd",l%," rt90"
```

--> Dessine trois losanges et un rhomboïde, un losange étant utilisé comme figure de départ et les autres figures étant obtenues par modification des graduations x ou y.

BOX x1,y1,x2,y2
PBOX x1,y1,x2,y2
RBOX x1,y1,x2,y2
PRBOX x1,y1,x2,y2

```
(x1,y1,x2,y2 : iexp)
```

BOX dessine un rectangle à partir des coordonnées x1,y1 et x2,y2 des coins opposés. PBOX dessine de la même façon un rectangle plein, RBOX un rectangle à coins arrondis et PRBOX un rectangle plein à coins arrondis.

Exemple :

```
BOX 20,20,120,120
RBOX 170,20,270,120
x=150
DEFFILL 1,2,4
PBOX 20,20+x,120,120+x
PRBOX 170,20+x,270,120+x
~INP(2)
```

--> Dessine un rectangle, puis un rectangle à coins arrondis et enfin ces deux mêmes figures à nouveau mais remplies. Pour terminer, le programme attend qu'une touche soit actionnée.

CIRCLE x,y,r [,a1,a2]
PCIRCLE x,y,r[,a1,a2]
ELLIPSE x,y,rx,ry[,a1,a2]
PELLIPSE x,y,rx,ry[,a1,a2]

```
(x,y,r,a1,a2 : iexp)
```

CIRCLE dessine un cercle de rayon r dont le centre est défini par les coordonnées x,y. Les angles de départ a1 et de fin a2 peuvent également être spécifiés pour définir un arc de cercle. PCIRCLE dessine de même un cercle plein ou une section de cercle pleine.

ELLIPSE dessine une ellipse autour du centre de coordonnées x,y avec un rayon horizontal rx et un rayon vertical ry. Les angles de départ a1 et de fin a2 peuvent également être spécifiés pour définir un arc d'ellipse. PELLIPSE dessine de même une ellipse pleine ou une section d'ellipse pleine. Les angles sont exprimés en 1/10 de degrés (0 à 3600) et les arcs ou sections de cercle sont dessinés dans le sens contraire des aiguilles d'une montre (c'est-à-dire 0 degrés droite, 90 degrés haut, 180 degrés gauche et 270 degrés bas).

Exemples :

```
CIRCLE 320,200,100
ELLIPSE 320,200,200,100,900,1800
PCIRCLE 320,200,100,1800,2700
PELLIPSE 320,200,200,100,2700,3600
~INP(2)
```

--> Dessine quelques (sections de) cercles et ellipses.

POLYLINE n,x(),y() [**OFFSET** off_x,off_y]
POLYMARK n,x(),y() [**OFFSET** off_x,off_y]
POLYFILL n,x(),y() [**OFFSET** off_x,off_y]

```
n,off_x,off_y : iexp
x(),y() : tableau avar
```

POLYLINE dessine une section de ligne avec n angles. Les coordonnées x,y des sommets d'angles sont définies dans les tableaux x() et y(). Le premier coin est défini par x(0) et y(0), le dernier par x(n-1) et y(n-1). Les premier et dernier coins sont automatiquement reliés entre eux. Un OFFSET (c'est-à-dire une distance) horizontal (off_x) ou vertical (off_y) peut être en outre additionné à ces coordonnées.

POLYFILL remplit le polygone avec un motif et une couleur qui peuvent être définis avec DEFFILL.

POLYMARK permet de marquer les points.

Exemple :

```
DIM x%(3),y%(3)
FOR i%=0 TO 3
  READ x%(i%),y%(i%)
NEXT i%
DATA 120,120,170,170,70,170,120,120
POLYLINE 4,x%(),y%()
POLYFILL 3,x%(),y%() OFFSET -50,-50
DEFMARK ,4,10
POLYMARK 3,x%(),y%() OFFSET 40,-80
~INP(2)
```

--> Dessine un triangle non rempli et un triangle plein ainsi que les marquages rectangulaires des coins d'un autre triangle.

POINT(x,y)

```
x,y : iexp
```

La couleur du point de coordonnés x,y est déterminée et renvoyée à la variable. Des valeurs entre 0 et 15 sont ici renvoyées en basse résolution, entre 0 et 3 en résolution moyenne et 0 ou 1 pour la haute résolution.

Exemple :

```
a=POINT(100,100)
PLOT 100,100
PRINT a,POINT(100,100)
```

--> Ecrit sur l'écran la couleur des coordonnées 100,100 de l'écran avant et après l'instruction PLOT.

FILL x,y [,c]

```
x,y,c : iexp
```

Cette instruction remplit une surface de configuration quelconque. L'opération de remplissage commence sur les coordonnées x,y. Si 'c' est spécifiée, l'opération de remplissage sera limitée par les points de couleur 'c' et par les limites de l'écran.

Si 'c' n'est pas spécifiée ou si c=-1, chaque point d'une autre couleur que le point de départ (x,y) sera considéré comme une limite.

Exemples :

```
LINE 0,180,639,180
FOR i=1 TO 19
  BOX i*20,100,i*20+20-i,180
  TEXT i*20-4,195,i
  DEFFILL ,2,i
  FILL i*20+1,101
NEXT i
```

--> Dessine une droite sur laquelle des rectangles de différents motifs se succèdent et remplit ces rectangles sans limite de couleur, ce qui détruit les motifs.

```
LINE 0,280,639,280
FOR i=1 TO 19
  BOX i*20,200,i*20+20-i,280
  TEXT i*20-4,295,i
  DEFFILL ,2,i
  FILL i*20+1,201,1
NEXT i
```

--> Dessine à nouveau, en dessous, une droite avec une série de rectangles de motifs différents qui sont ensuite remplis avec limitation par une couleur. Les différents motifs sont ainsi préservés.

CLS [#n]

```
n : iexp
```

Vide l'écran en sortant ESC-E-CR. Peut aussi être dirigé sur des fichiers.

Exemple :

```
PBOX 100,100,500,200
REPEAT
UNTIL MOUSEK
CLS
```

--> Remplit une partie de l'écran avec un rectangle. L'écran se vide ensuite dès qu'un bouton de la souris est actionné.

TEXT x,y [,l] ,expression

```
x,y,l : iexp
expression : sexp ou aexp
```

Sort l'expression sous forme de texte graphique à partir du point de coordonnées x,y. Ce point est défini par rapport au coin gauche de la ligne de base du premier caractère de *expression*. Le paramètre l fixe la longueur de la sortie de texte en points écran. Cette longueur est définie en modifiant l'espacement des caractères (pour l>0) ou bien en modifiant l'espacement des mots (pour l<0). Si l=0, expression est sortie sans modification.

Le texte graphique peut être doté d'attributs à l'aide de DEFTEXT. DEFTEXT n'a cependant d'effet que sur les instructions TEXT et PRINT à l'intérieur d'une même "fenêtre".

Exemple :

```
s$="Voici un exemple."
FOR i=0 TO 23
  DEFTEXT 1,i,0,6
  TEXT 50,i*16+16,s$
NEXT i
DEFTEXT 1,0,0,13
TEXT 350,50,s$
TEXT 350,100,s$
TEXT 350,150,250,s$
```

```
TEXT 350,200,-250,s$
~ INP(2)
```

--> Ecrit le texte graphique sur l'écran avec diverses variantes puis attend qu'une touche soit actionnée.

SPRITE motif_bits$ [,x,y]

```
(motif_bits$ : svar)
```

L'instruction SPRITE permet de déplacer sur l'écran un objet graphique d'une taille de 16 points écran sur 16. Les informations de bits appropriées pour le motif et le masque sont placées dans une chaîne de caractères. Il convient de noter à cet égard que toutes ces valeurs doivent être fournies sous forme de mots. On peut utiliser à cet effet l'instruction MKI$, de sorte que *motif_bits$* pourra être ainsi constitué :

```
motif_bits$=MKI$(coordonnée x du point d'action)
                +MKI$(coordonnée y du point d'action)
                +MKI$(0) pour normal ou MKI$(-1) pour XOR
                +MKI$(couleur du masque) généralement 0
                +MKI$(couleur du sprite) généralement 1
                +sprite$
```

sprite$ contient ici les informations de bits pour la forme et le masque du sprite qui ne doivent pas être indiquées successivement, comme pour DEFMOUSE, mais alternativement.

Exemple :

```
gfa$=MKI$(1)+MKI$(1)+MKI$(0)
gfa$=gfa$+MKI$(0)+MKI$(1)
FOR i%=1 TO 16
  READ motif%,masque%
  gfa$=gfa$+MKI$(masque%)+MKI$(motif%)
NEXT i%
DATA 0,0,0,32256,15360,16896,8192,24064
DATA 8192,24560,11744,21008,9472,23280,9472,23295
DATA 15838,16929,274,32493,274,749,286,737
DATA 18,1005,18,45,18,45,0,63
REPEAT
  ADD mx%,(MOUSEX-mx%)/50
  ADD my%,(MOUSEY-my%)/50
  SPRITE gfa$,mx%,my%
UNTIL MOUSEK=2
```

--> Déplace sur l'écran un sprite qui suit à distance le curseur de la souris.

SECTIONS D'ECRAN

SGET ecran$
SPUT ecran$

```
ecran$ : svar
```

SGET copie l'écran tout entier (32000 octets) dans une chaîne de caractères, de façon particulièrement rapide. SPUT copie de même une chaîne de 32000 caractères (=octets) dans la mémoire écran.

```
PCIRCLE 100,100,50
SGET b$
~INP(2)
CLS
~INP(2)
SPUT b$
```

--> Dessine une cercle plein sur l'écran. Ce cercle disparaît dès qu'une touche est actionnée puis réapparaît si une touche est à nouveau actionnée.

GET x1,y1,x2,y2,section$
PUT x1,y1,section$ [,mode]

```
(x1,y1,x2,y2,mode : iexp)
(section$ : svar)
```

GET stocke une section de l'écran dans une variable de chaîne. PUT envoie de même une section de l'écran stockée avec GET dans l'emplacement x,y de la mémoire écran. *mode* permet, en option, de définir comment le motif de bits défini dans *section$* doit être combiné avec le contenu actuel de l'écran. Dans la table suivante, S (Source) désigne le motif de bits contenu dans la chaîne et D (Destination) le contenu originel de l'écran.

Mode	Formule de combinaison	Effet
0	0	Tous les points sont effacés
1	S AND D	Seuls restent fixés les points fixés dans les deux grilles.
2	S AND (NOT D)	Seuls sont fixés les points fixés dans la grille source mais effacés dans la grille de destination.
3	S	La grille source est transférée sans modification (GRAPHMODE 1, valeur prédéfinie)
4	(NOT S) AND D	Seuls sont fixés les points fixés dans la grille source et fixés dans la grille de destination.
5	D	La destination n'est pas modifiée.
6	S XOR D	Seuls sont fixés les points fixés dans une seule des grilles (GRAPHMODE 3).
7	S OR D	Sont fixés tous les points fixés dans l'une des deux grilles (GRAPHMODE 2).
8	NOT (S OR D)	Sont fixés tous les points qui ne sont fixés dans aucune des deux grilles.
9	NOT (S XOR D)	Sont fixés tous les points fixés dans les deux grilles ou bien dans aucune d'entre elles.
10	NOT D	La grille de destination est inversée.
11	S OR (NOT B)	Sont fixés tous les points fixés dans la grille source ainsi que tous ceux qui n'étaient pas fixés dans la grille de destination.
12	NOT B	La grille source est inversée avant transfert.
13	(NOT S) OR B	GRAPHMODE 4
14	NOT (S AND D)	Sont fixés tous les points qui ne figuraient pas dans les deux grilles à la fois.
15	1	Tous les points sont fixés.

Si en outre le bit 4 du mode est fixé, le motif de remplissage subira une combinaison supplémentaire AND avec la grille source.

VSYNC

Cette instruction sert à synchroniser la construction de l'écran (c'est-à-dire que le programme attend le retour vertical du faisceau du moniteur *'vertical blank interrupt'*).

VSYNC peut être employé par exemple pour l'animation de sections de l'écran avec GET et PUT.

Exemple :

```
t%=TIMER
FOR i%=1 TO 100
  VSYNC
NEXT i%
PRINT SUB(TIMER,t%)/200
```

--> Affiche la durée de 100 changements d'image.

BITBLT s_mfdb%(),d_mfdb%(),par%()

```
s_mfdb%(),d_mfdb%(),par%() : tableaux entiers
```

L'instruction BITBLT sert à copier des sections rectangulaires de l'écran. Elle ressemble aux instructions GET et PUT mais elle est plus souple et plus rapide malgré son emploi plus complexe.

Les paramètres de cette instruction sont placés dans trois tableaux dont le premier décrit la structure de la zone à copier (grille source), le second la zone où la copie doit être placée (grille de destination) et le troisième contient, enfin, les coordonnées des zones source et de destination ainsi que le mode de copie.

Cette instruction repose sur une routine VDI alors que BITBLT adr% et BITBLT x%() emploient une routine LINE-A (voyez la section sur les appels LINE-A).

La structure de la grille source (s_mfdb%) et de la grille de destination (d_mfdb%) obéissent aux même règles. Voici la signification de ces abréviations :

```
s_mfdb%() source memory form description block
d_mfdb%() destination memory form description block
```

Les éléments du tableau sont :

```
_mfdb%(0)   Adresse de départ de la grille. Cette adresse doit être paire.
_mfdb%(1)   Largeur de la grille en points écran. Cette valeur doit être
            divisible par 16.
_mfdb%(2)   Hauteur de la grille en points écran.
_mfdb%(3)   Largeur de la grille en mots (= nombre de points écran / 16)
_mfdb%(4)   Réservé, toujours 0
_mfdb%(5)   Nombre de niveaux de grille
            haute résolution = 1
            résolution moyenne = 2
            résolution basse = 4
```

Le tableau par%() présente la structure suivante :

```
par%(0)     Coordonnée gauche x de la grille source
par%(1)     Coordonnée supérieure y de la grille source
par%(2)     Coordonnée droite x de la grille source
par%(3)     Coordonnée inférieure y de la grille source
par%(4)     Coordonnée gauche x de la grille de destination
par%(5)     Coordonnée supérieure y de la grille de destination
par%(6)     Coordonnée droite x de la grille de destination
par%(7)     Coordonnée inférieure y de la grille de destination
par%(8)     Mode de copie
```

Les valeurs du mode de copie sont les mêmes que pour GET/PUT (voyez aussi le chapitre consacré au graphisme). Voici les principales définitions :

```
  3 = replace              (GRAPHMODE 1)
  6 = xor                  (GRAPHMODE 2)
  7 = transparent          (GRAPHMODE 3)
 13 = invers transparent   (GRAPHMODE 4)
```

Exemple :

```
DIM smfdb%(8),dmfdb%(8),p%(8)
'
FOR i%=0 TO 639 STEP 8
  CIRCLE i%,i%,399
NEXT i%
'
GET 0,0,639,399,a$
mirrorput(0,0,a$)
'
PROCEDURE mirrorput(x%,y%,VAR x$)
  IF LEN(x$)>6          ! Seulement s'il y a bien quelque chose
    a%=V:x$
    b%=INT(a%)
    h%=INT(a%+2)
    '
    smfdb%(0)=a%+6
    smfdb%(1)=(b%+16) AND &HFFF0
    smfdb%(2)=h%+1
    smfdb%(3)=smfdb%(1)/16
    smfdb%(5)=DPEEK(a%+4)
    dmfdb%(0)=XBIOS(3)
    '
    dmfdb%(1)=640
    dmfdb%(2)=400
    dmfdb%(3)=40
```

```
      dmfdb%(5)=1
      '
      p%(1)=0
      p%(3)=h%
      p%(5)=y%
      p%(7)=y%+h%
      p%(8)=3
      p%(4)=x%+b%
      p%(6)=x%+b%
      '
      FOR i%=0 TO b%
        p%(0)=i%
        p%(2)=i%
        BITBLT smfdb%(),dmfdb%(),p%()
        DEC p%(4)
        DEC p%(6)
      NEXT i%
      '
    ENDIF
  RETURN
```

--> Dessine une succession de cercles. L'écran tout entier est ensuite chargé dans une chaîne de caractères puis reflété par rapport à l'axe vertical (cf. BITBLT dans la section sur les appels LINE-A).

9. GESTION DES EVENEMENTS, MENUS ET FENETRES

GESTION DES EVENEMENTS

Un certain nombre d'instructions caractérisques du GFA-BASIC permettent de tester un certain nombre d'événements simples (qu'on appelle *Events* dans la littérature consacrée à GEM). Ces événements sont le fait d'appuyer sur un bouton de la souris, le fait d'appuyer sur une touche, la surveillance des coordonnées de la souris par rapport à deux zones rectangulaires de l'écran et l'intervention d'un "message GEM" qui peut contenir par exemple des informations sur la gestion des fenêtres.

La surveillance de ces événements est activée à l'aide d'un ON MENU xxx GOSUB où xxx représente l'événement auquel il s'agit de réagir. La surveillance elle-même s'effectue à l'aide de l'instruction ON MENU. Chaque fois que cette instruction est appelée, on teste si un événement s'est produit. Si un événement auquel il faut réagir est effectivement intervenu, on saute à la procédure prévue à cet effet.

ON MENU [t]

```
t : iexp
```

L'instruction ON MENU surveille l'intervention d'événements. Avant cette instruction, il convient de définir à quelle procédure le programme devra sauter lors de tel ou tel événement. Les instructions permettant de le définir seront décrites dans la suite de ce chapitre. Pour qu'un événement soit surveillé en permanence, il est nécessaire de répéter sans cesse cette instruction. C'est pourquoi l'instruction ON MENU est normalement placée dans une boucle.

Le paramètre t contient le temps (en millièmes de seconde) au bout duquel l'instruction ON MENU doit être terminée. Cette indication de durée peut se révéler utile dans la mesure où GEM ne remarque pas toujours qu'un bouton de la souris a été relâché (effet caractéristique : le champ de fermeture d'une fenêtre est inversé mais la fenêtre reste ouverte jusqu'à ce qu'on clique plusieurs fois énergiquement). C'est ce que cette variante de l'instruction devrait permettre d'éviter, à condition toutefois que les boutons de la souris soient effectivement surveillés, ce qui nécessite une instruction ON MENU BUTTON x,y,z GOSUB.

Exemple :

```
ON MENU BUTTON 1,1,1 GOSUB test
t%=TIMER
REPEAT
  PRINT (TIMER-t%)/200
  ON MENU 2000
UNTIL MOUSEK=2
PROCEDURE test
RETURN
```

--> Affiche le temps écoulé depuis le lancement du programme. Si le bouton gauche de la souris est actionné, alors l'événement est détecté et le temps écoulé n'est plus affiché que toutes les deux secondes. Le fait d'appuyer sur le bouton droit de la souris entraîne l'arrêt du programme.

MENU(x)

```
(x : aexp entre -2 et 15 inclus)
```

Les variables MENU(-2) à MENU(15) contiennent toutes les informations nécessaires à la gestion des événements. Lorsqu'une entrée d'un menu déroulant a été sélectionnée, MENU(0) contient l'index de l'entrée sélectionnée dans le tableau des entrées du menu (voyez la prochaine section ainsi que le programme d'exemple à la fin de la section sur les menus déroulants).

MENU(-2) contient l'adresse du buffer de message et MENU(-1) l'adresse de l'arbre des objets du menu déroulant.

Les variables MENU(1) à MENU(8) contiennent le buffer de message et MENU(9) à MENU(15) le AES Integer Output Block (bloc de sortie des entiers AES). Nous allons maintenant évoquer brièvement l'utilisation des informations figurant à ces endroits.

Le point de départ de ces explications sera MENU(1) et le buffer de message. MENU(1) contient le numéro de code de l'événement intervenu. En fonction de MENU(1), les autres éléments du buffer de message contiendront différentes informations que la table suivante énumère. La table présente chaque fois la valeur de MENU(1) suivie de la signification de cette valeur et enfin des variables dans lesquelles figurent des informations importantes pour cette valeur de MENU(1). Ces informations importent avant tout pour permettre une gestion correcte des fenêtres :

```
MENU(1) = 10 :    une entrée du menu déroulant a été sélectionnée
     MENU(0) :    index de l'entrée du menu dans le tableau des entrées
     MENU(4) :    index d'objet du titre du menu
```

```
     MENU(5) :   index d'objet de l'entrée du menu sélectionnée
MENU(1) = 20 :   une zone rectangulaire de la fenêtre doit être redessinée
     MENU(4) :   code (Handle) de la fenêtre
     MENU(5) :   coordonnée gauche X de la zone
     MENU(6) :   coordonnée supérieure Y de la zone
     MENU(7) :   largeur de la zone
     MENU(8) :   hauteur de la zone
                 (Exemple sous ON MENU MESSAGE GOSUB)

MENU(1) = 21 :   une fenêtre a été cliquée (cela signifie normalement que
                 l'utilisateur veut activer cette fenêtre)
     MENU(4) :   code (Handle) de la fenêtre cliquée

MENU(1) = 22 :   le champ de fermeture d'une fenêtre a été cliqué
     MENU(4) :   code (Handle) de cette fenêtre

MENU(1) = 23 :   le champ en haut à droite d'une fenêtre a été cliqué (cela
                 signifie normalement que l'utilisateur veut déployer la
                 fenêtre à la taille maximale)
     MENU(4) :   code (Handle) de cette fenêtre

MENU(1) = 24 :   un des quatre champs fléchés ou un champ de bouton-poussoir du
                 cadre de la fenêtre a été cliqué. Le fait de déplacer un bouton
                 entraine MENU(1)=25 ou 26. On teste ici simplement si on a cliqué
                 dans la zone dessinée à droite ou à gauche de la position actuelle
                 du bouton-poussoir.
     MENU(4) :   code (Handle) de la fenêtre
     MENU(5) :   Où a-t-on cliqué ? Les conventions suivantes s'appliquent :
           0 :   au-dessus du bouton de droite
           1 :   en dessous du bouton de droite
           2 :   flèche vers le haut
           3 :   flèche vers le bas
           4 :   à gauche du bouton inférieur
           5 :   à droite du bouton inférieur
           6 :   flèche vers la gauche
           7 :   flèche vers la droite

MENU(1) = 25 :   le bouton-poussoir inférieur a été déplacé
     MENU(4) :   code (Handle) de la fenêtre
     MENU(5) :   position du bouton (nombre entre 1 et 1000)

MENU(1) = 26 :   le bouton-poussoir droit a été déplacé
     MENU(4) :   code (Handle) de la fenêtre
     MENU(5) :   position du bouton-poussoir (nombre entre 1 et 1000)
```

```
MENU(1) = 27 :   la taille de la fenêtre a été modifiée à l'aide de l'élément
                 utilitaire en bas à droite. La nouvelle taille de la fenêtre
                 est renvoyée dans :
     MENU(4) :   code (Handle) de la fenêtre
     MENU(5) :   coordonnée gauche X
     MENU(6) :   coordonnée supérieure Y
     MENU(7) :   largeur de la fenêtre
     MENU(8) :   hauteur de la fenêtre

MENU(1) = 28 :   la position d'une fenêtre a été modifiée, les nouvelles
                 coordonnées figurent dans :
     MENU(4) :   code (Handle) de la fenêtre
     MENU(5) :   coordonnée gauche X
     MENU(6) :   coordonnée supérieure Y
     MENU(7) :   largeur de la fenêtre
     MENU(8) :   hauteur de la fenêtre

MENU(1) = 29 :   une nouvelle fenêtre a été activée par GEM.
     MENU(4) :   code (Handle) de la fenêtre

MENU(1) = 40 :   un accessoire a été sélectionné. Cette valeur ne peut être reçue
                 que par un accessoire qui doit tester s'il a été appelé d'après
                 la valeur figurant dans MENU(4).
     MENU(4) :   numéro d'identification de l'accessoire dans le menu

MENU(1) = 41 :   un accessoire a été refermé. Cette valeur ne peut être reçue que
                 par un accessoire qui doit tester s'il a été terminé d'après la
                 valeur figurant dans MENU(4).
     MENU(4) :   numéro d'identification de l'accessoire dans le menu
```

Les variables MENU(9) à MENU(15) et GINTOUT(0) à GINTOUT(7) contiennent les informations suivantes si le bit correspondant à l'événement approprié est mis dans MENU(9) :

MENU(9) indique dans un bit correspondant à chaque événement le type de l'événement intervenu, d'après les conventions suivantes :

```
Bit 0 --> clavier
Bit 1 --> bouton de la souris
Bit 2 --> souris vient de quitter/pénétrer dans rectangle 1
Bit 3 --> souris vient de quitter/pénétrer dans rectangle 2
Bit 4 --> un message vient d'arriver dans le buffer de message
Bit 5 --> temporisateur (timer)

MENU(10) :  position X de la souris lors de l'intervention de l'événement
MENU(11) :  position Y de la souris lors de l'intervention de l'événement
```

```
MENU(12) :  état des boutons de la souris, d'après les conventions suivantes :
            0 --> aucun bouton actionné
            1 --> bouton gauche actionné
            2 --> bouton droit actionné
            3 --> deux boutons actionnés
            (Exemple sous ON MENU BUTTON x,y,z GOSUB)

MENU(13) :  état des touches de commutation du clavier ; un bit est fixé pour
            chaque touche de commutation actionnée, d'après les conventions
            suivantes :
            Bit 0 --> touche Shift de droite
            Bit 1 --> touche Shift de gauche
            Bit 2 --> touche Control
            Bit 3 --> touche Alternate
            (Exemple sous ON MENU KEY GOSUB)

MENU(14) :  informations sur la touche actionnée. Dans l'octet du bas figure le
            code ASCII de la touche actionnée, dans l'octet du haut le code
            clavier (exemple sous ON MENU KEY GOSUB).

MENU(15) :  Nombre de clics de la souris ayant entrainé l'événement.
```

ON MENU BUTTON clics,bouton,etat GOSUB proc

```
(clics,bouton,etat : iexp)
(proc : nom d'une procédure)
```

Cette instruction sert à surveiller les boutons de la souris. Les variables *clics*, *bouton* et *etat* définissent quel état des boutons de la souris devra entraîner un saut à la procédure *proc*.

clics indique le nombre maximal de clics des boutons de la souris à enregistrer. *bouton* indique la combinaison des boutons attendue, d'après les conventions suivantes :

```
0 --> aucun bouton
1 --> bouton gauche
2 --> bouton droit
3 --> les deux boutons en même temps
```

L'expression *etat* indique l'état que doivent présenter les boutons ainsi spécifiés. 0 signifie ici un bouton non enfoncé et 1 un bouton enfoncé. Le nom *proc* indique à quelle procédure il faudra sauter lorsque l'événement défini par *clics*, *bouton* et *etat* se sera produit.

La surveillance s'effectue lors de chaque instruction ON MENU.

Exemple :

```
ON MENU BUTTON 1,1,0 GOSUB box
GRAPHMODE 3
REPEAT
  ON MENU
UNTIL MOUSEK=2
'
PROCEDURE box
  ADD i%,7
  IF i%>200
    i%=3
  ENDIF
  BOX 320-i%,200-i%,320+i%,200+i%
RETURN
```

--> Dessine un jeu graphique sur l'écran après avoir cliqué sur le bouton gauche de la souris. Le fait d'appuyer sur le bouton droit met fin au programme.

ON MENU KEY GOSUB proc

```
proc : nom d'une procédure
```

Cette instruction met en place une surveillance du clavier. *proc* est le nom d'une procédure à laquelle il faudra sauter lorsqu'une instruction ON MENU constatera qu'une touche aura été actionnée.

```
ON MENU KEY GOSUB evaluation_touche
REPEAT
  ON MENU
UNTIL MOUSEK=2
'
PROCEDURE evaluation_touche
  PRINT "Touches de commutation du clavier : ";MENU(13)
  PRINT "Code ASCII : ";BYTE(MENU(14))
  PRINT "Code clavier : ";SHR(MENU(14),8)
  PRINT
RETURN
```

--> Lorsqu'une touche est actionnée l'état actuel des touches de commutation du clavier *(Shift, Control, Alternate)* est annoncé ainsi que les codes ASCII et clavier de la touche actionnée. Le fait d'appuyer sur le bouton droit de la souris met fin au programme. Sur la signification de MENU(13) et MENU(14), voyez sous MENU(x).

ON MENU IBOX no,x,y,l,h **GOSUB** proc
ON MENU OBOX no,x,y,l,h **GOSUB** proc

(no,x,y,l,h : iexp)
(proc : nom d'une procédure)

Ces deux instructions surveillent les coordonnées de la souris. On saute à la procédure *proc* lorsque la souris pénètre dans (IBOX) ou sort (OBOX) d'une zone rectangulaire de l'écran.

Il est possible de définir à cet effet deux zones rectangulaires de l'écran qui seront surveillées indépendamment l'une de l'autre. no est le numéro du rectangle correspondant, x sa coordonnée gauche X, y sa coordonnée supérieure Y, l la largeur et h la hauteur du rectangle.

La surveillance s'effectue lors de l'exécution d'une instruction ON MENU.

Exemple :

```
ON MENU IBOX 1,250,130,140,140 GOSUB entre_dans_boite
ON MENU OBOX 2,50,50,540,300 GOSUB sort_de_boite
'
GRAPHMODE 3
BOX 250,130,390,270
BOX 50,50,590,350
REPEAT
  ON MENU
UNTIL MOUSEK=2
'
PROCEDURE entre_dans_boite
  BOX 250+i%,130+i%,390-i%,270-i%
  IF i%=70
    i%=2
  ENDIF
  ADD i%,2
RETURN
'
PROCEDURE sort_de_boite
  BOX 0+j%,0+j%,639-j%,399-j%
  IF j%=0
    j%=50
  ENDIF
  SUB j%,2
RETURN
```

--> Si la souris pénètre dans le rectangle intérieur, un jeu graphique s'y développe. Si elle quitte le rectangle extérieur, c'est dans ce rectangle que ce jeu se déroule. Le fait d'appuyer sur le bouton droit de la souris met fin au programme.

ON MENU MESSAGE GOSUB proc

(proc : nom d'une procédure)

Lorsqu'un message parvient dans le buffer de message, le programme saute à la procédure dénommée *proc*. La surveillance du buffer de message s'effectue lors de chaque instruction ON MENU. La structure du buffer de message est décrite dans la section sur MENU(x).

Exemple :

```
DIM m$(10)
FOR i%=0 TO 10
  READ m$(i%)
NEXT i%
DATA Desk, Redraw ,------------------------
DATA 1,2,3,4,5,6,"","",""
OPENW 0
MENU m$()
ON MENU MESSAGE GOSUB evaluation_message
PRINT AT(1,1);
REPEAT
  ON MENU
UNTIL MOUSEK=2
'
PROCEDURE evaluation_message
  IF MENU(1)=20
    PRINT CHR$(7)
    PRINT "Une section de l'écran"
    PRINT "doit être redessinée."
  ELSE
    PRINT CHR$(7)
    PRINT "Il s'est passé quelque chose !!!"
  ENDIF
RETURN
```

--> Ce programme crée des accessoires. Si vous en sélectionnez un, le programme remarquera qu'une section de l'écran a été recouverte et il vous l'annoncera (voyez aussi MENU(x) à ce propos). Le programme peut être terminé en appuyant sur le bouton droit de la souris.

MENUS DEROULANTS

Cette section décrit les instructions spécifiques de GFA-BASIC 3.0 pour la gestion des menus déroulants. Il règne malheureusement une certaine confusion dans les termes dans la littérature consacrée à ce thème. C'est pourquoi il nous faut tout d'abord préciser la signification des termes que nous emploierons en la matière dans ce manuel. Nous utiliserons le terme de menu déroulant comme terme générique. Dans la ligne la plus haute de l'écran figure la partie du menu déroulant qui reste toujours visible, la ligne ou barre des menus. Cette ligne contient les différents titres des menus. Lorsque la flèche de la souris passe sur l'un de ces titres, un menu se déroule sous ce titre. Chaque élément de ce menu, qui peut être sélectionné individuellement, constitue ce qu'on appelle une entrée du menu. Les termes ainsi choisis n'ont pas de caractère normatif et d'autres ouvrages appliquent d'autres définitions.

Lorsqu'un menu déroulant est mis en place, ces entrées sont définies dans un tableau de chaînes m$(). L'instruction MENU m$() envoie ce menu déroulant sur l'écran. L'instruction ON MENU GOSUB définit une procédure à laquelle le programme devra sauter après qu'une entrée du menu aura été sélectionnée. Pendant le déroulement du programme, chaque instruction ON MENU fait tester si une entrée a été sélectionnée.

L'instruction MENU OFF affiche à nouveau en écriture normale les entrées inversées de la ligne du menu. MENU KILL désactive le menu déroulant.

L'instruction MENU(x,y) permet de doter des entrées du menu de crochets ou de les faire afficher en caractères clairs, ce qui entraîne également que l'entrée du menu correspondante ne pourra plus être sélectionnée.

ON MENU GOSUB proc
MENU m$()

```
(proc : nom d'une procédure)
(m$() : tableau de chaines)
```

Ces deux instructions se chargent de la mise en place et de la gestion d'un menu déroulant. Elles s'appuient pour ce faire sur les instructions et variables décrites à la section précédente (ON MENU, MENU()). ON MENU GOSUB proc définit la procédure à laquelle il faudra sauter lorsqu'une entrée du menu aura été sélectionnée. Si cette entrée est un accessoire, cette procédure ne sera pas appelée. A l'intérieur de la procédure, la variable MENU(0) permet de rechercher quelle entrée du menu a été sélectionnée. MENU(0) est l'index de l'entrée sélectionnée dans le

tableau des entrées m$() ; m$(MENU(0)) représente alors le texte cliqué dans le menu déroulant si c'est OPTION BASE 0 qui s'applique. Si OPTION BASE 1 a été sélectionnée, le texte de l'entrée sélectionnée figure dans m$(MENU(0)+1).

MENU m$() affiche le menu déroulant sur l'écran. Dans le tableau de chaînes m$() figurent les titres, les entrées et les emplacements réservés pour les accessoires. Le format suivant doit être respecté pour disposer les entrées dans le tableau m$:

m$(0)	Titre du menu, qui peut comporter des accessoires.
m$(1)	Nom de la première entrée du premier menu.
m$(2)	Une séquence de signes moins (-).
m$(3) à m$(8)	Emplacements réservés pour les accessoires. Il s'agit simplement de six chaînes de caractères d'une longueur supérieure à zéro. Si des accessoires ont été chargés lors de la mise en marche de l'ordinateur, ils apparaissent dans le menu déroulant sous leur nom, sinon les entrées correspondantes du menu ne sont pas affichées.
m$(9)	Une chaîne vide marquant la fin du premier menu.

Tous les autres menus présentent le format suivant :

1. Titre du menu
2. Liste des entrées du menu
3. Une chaîne vide marquant la fin du premier menu

Le dernier menu est encore suivi d'une autre chaîne vide qui marque la fin du menu déroulant dans son ensemble. Une entrée de menu commençant par un signe *moins* est automatiquement affichée en clair et ne peut être sélectionnée.

Exemple :

voyez la fin de la section.

MENU OFF
MENU KILL

Lorsqu'une entrée du menu est sélectionnée, le titre de ce menu est inversé. MENU OFF rétablit l'état normal du titre.

MENU KILL désactive le menu déroulant qui n'est toutefois pas effacé de l'écran. MENU KILL annule en outre les réglages opérés avec ON MENU GOSUB.

MENU x,y

```
(x,y : aexp)
```

Cette instruction permet de changer l'état de l'entrée numéro x d'un menu déroulant. La numérotation des entrées correspond aux indices du tableau de chaînes dans lequel figurent les entrées du menu déroulant. On compte donc à partir de zéro. Les titres, les entrées fictives pour les accessoires et les chaînes vides sont pris en compte dans cette numérotation.

Le second paramètre y définit ce qu'il doit advenir de l'entrée numéro x du menu, d'après les conventions suivantes :

y		Effet
0	-->	pas de crochet devant
1	-->	fixer un crochet devant
2	-->	caractères clairs, ne peut être sélectionnée
3	-->	caractères normaux, entrée peut être sélectionnée

Exemple :

voyez la fin de la section.

Programme d'exemple pour la section consacrée aux menus déroulants :

```
Dim Entree$(20)
I%=-1
Repeat
  Inc I%
  Read Entree$(I%)
Until Entree$(I%)="Marque de fin"
Entree$(I%)=""
'
Menu Entree$()
On Menu  Gosub Evaluation
Openw 0
'
Data Desk,  Test ,--------------------,1,2,3,4,5,6,
Data File,  Load ,  Save ,-------,  Quit ,
Data Titre,  Entrée 1 ,  Entrée 2 ,
Data Marque de fin
'
Repeat
```

```
  On Menu
Until Mousek=2
'
Procedure Evaluation
  Menu Off
  ' menu(0) contient l'index de l'entrée sélectionnée
  ' dans le tableau
  M%=Menu(0)
  Print Entree$(M%)
  '
  Alert 0,"Indicateur devant ?",1," Oui | Non ",A%
  If A%=1
    Menu M%,1
  Else
    Menu M%,0
  Endif
  '
  Alert 0,"Caractères clairs ?|(ne peut être sélectionné)",1," Oui | Non ",A%
  If A%=1
    Menu M%,2
  Else
    Menu M%,3
  Endif
Return
```

--> Met en place et gère un menu déroulant. Lorsqu'une entrée du menu est cliquée, son texte est écrit sur le moniteur et on vous demande comment l'entrée du menu sélectionnée doit apparaître. La sélection de "Caractères clairs" fait passer l'option sélectionnée en grisé dans le menu et interdit que celle-ci soit à nouveau choisie.

INSTRUCTIONS DE FENETRES

GFA-BASIC offre quelques instructions simples pour la gestion des fenêtres qui ont toutefois l'inconvénient de ne pas être très souples (OPENW, CLOSEW, CLEARW, TITLEW, INFOW). Pour programmer une gestion des fenêtres plus puissante, il faut avoir recours aux routines AES appropriées ou bien placer les paramètres de la fenêtre voulue dans une table (WINDTAB) et appeler à nouveau la fenêtre. Vous disposez en outre des fonctions W_HAND et W_INDEX qui servent de pont entre les instructions simples de GFA-BASIC et les fonctions AES plus puissantes de la bibliothèque des fenêtres.

Les instructions élémentaires sont basées sur un nombre réduit de positions variables des fenêtres. Un coin de la fenêtre se situera toujours dans la même position alors que le coin opposé sera placé sur le point de contact des quatre fenêtres.

Lorsque les instructions élémentaires sont utilisées, l'origine du système de coordonnées pour les sorties de texte et les sorties graphiques est placé sur le coin supérieur gauche de la fenêtre. Cela ne s'applique cependant pas à certaines instructions, par exemple à GET et PUT, BITBLT etc... Les sorties graphiques et les sorties de texte qui débordent des limites de la fenêtre sont automatiquement tronquées (Clipping).

OPENW #n,x,y,w,h,attr
OPENW no [,pos_x,pos_y]
CLOSEW [#] no

```
no,pos_x,pos_y : aexp
n,x,y,w,h,attr : iexp
```

OPENW ouvre la fenêtre de numéro no. Les paramètres *pos_x* et *pos_y* définissent la position d'un coin de la fenêtre de façon fixe. Le coin opposé, en diagonale, est également fixe. Seules les routines AES ou WINDTAB permettent une gestion plus souple des fenêtres. Les coordonnées (x,y) des fenêtres possibles sont :

No	En haut à gauche	En bas à droite
1	(0,19)	(pos_x,pos_y)
2	(pos_x,19)	(639,pos_y)
3	(0,pos_y)	(pos_x,399)
4	(pos_x,pos_y)	(639,399)

Le point (pos_x,pos_y) est donc le point de contact entre les quatre fenêtres.

L'instruction OPENW 0 n'ouvre pas une véritable fenêtre mais déplace simplement l'origine des coordonnées vers le point (0,19). Les 19 lignes du haut de l'écran deviennent ainsi inaccessibles aux instructions de sortie de texte ou de sortie graphique. Cela peut par exemple servir à protéger une ligne de menu déroulant contre tout empiètement.

L'instruction CLOSEW ferme la fenêtre de numéro no, CLOSEW # la fenêtre d'index n.

Exemple :

(voyez aussi le programme d'exemple à la fin de la section).

```
REPEAT
  IF MOUSEK=1
    CLOSEW 1
```

```
    OPENW 4,320,200
  ENDIF
  IF MOUSEK=2
    CLOSEW 4
    OPENW 1,100,100
  ENDIF
UNTIL MOUSEK=3
CLOSEW #1
CLOSEW #4
```

--> Le fait d'appuyer sur le bouton gauche de la souris ouvre la fenêtre 1, le fait d'appuyer sur le bouton droit de la souris ouvrant la fenêtre 4. Le fait d'appuyer sur les deux boutons simultanément met fin au programme.

La seconde variante de OPENW ouvre la fenêtre d'index n sur la position x,y avec la largeur w et la hauteur h. L'expression *attr* définit quels éléments la fenêtre doit comporter (voyez aussi WIND_CREATE dans la section sur l'AES).

Exemple :

```
TITLEW #1,"Titre"                          ! Ouvre la fenêtre d'index 1
INFOW # 1,STRING$(15,"....|")              ! Prédéfinir la ligne
'                                            d'informations
OPENW #1,16,32,600,300,&X111111111111      ! Fixer coordonnées et
'                                            attribut
-INP(2)
CLOSEW #1                                  ! Très important !!, fermer
'                                            fenêtre
```

--> Ouvre une fenêtre avec lignes de titre et d'informations. Le fait d'appuyer sur une touche met fin au programme.

W_HAND(#n)
W_INDEX(#hd)

```
(n, hd : aexp)
```

W_HAND renvoie le handle GEM de la fenêtre dont le code a été indiqué dans n. W_INDEX représente la fonction inverse et renvoie le numéro de fenêtre correspondant au handle GEM spécifié.

Exemple :

```
OPENW 2
PRINT W_HAND(#2)
```

```
~ INP(2)
CLOSEW #2
```

--> Ecrit le handle de la fenêtre de code 2 sur l'écran. Le fait d'appuyer sur une touche met fin au programme.

CLEARW [#] no
TITLEW [#] no,a$
INFOW [#] no,a$
TOPW # no
FULLW [#] no

L'instruction CLEARW vide la fenêtre numéro no. TITLEW écrit le texte a$ dans la ligne la plus haute de la fenêtre. INFOW écrit le texte a$ dans la ligne d'information, placée juste en dessous, de la fenêtre no. TOPW active la fenêtre numéro no. FULLW déploie la fenêtre no sur toute la taille de l'écran.

Exemple :

(voyez aussi le programme d'exemple à la fin de la section).

```
DEFFILL 1,2,4
PBOX 0,0,639,399
OPENW 1
PAUSE 50
FULLW #1
PRINT "Fenêtre 1"
OPENW 4,100,100
PAUSE 50
CLEARW 1
OPENW 3
PAUSE 50
TOPW #1
PAUSE 50
CLOSEW #1
TITLEW 4,"Fenêtre 4"
INFOW 3,"Fenêtre 3"
PAUSE 100
CLOSEW #3
CLOSEW 4
```

--> Fait apparaître quelques fenêtres, les modifie puis les démantèle.

WINDTAB
WINDTAB(i,j)

```
i,j : iexp
```

A partir de l'adresse figurant dans WINDTAB figure l'adresse de la table de paramètres dans laquelle se trouvent toutes les informations définissant l'apparence d'une fenêtre. L'origine des coordonnées pour les sorties graphiques peut également être fixée dans cette table.

Cette table comporte 68 octets organisés en mots (2 octets représentant chaque fois une entrée de la table). L'emploi de la table est expliqué à la fin de la section dans un programme d'exemple. Ce programme inscrit les paramètres de la fenêtre à mettre en place dans la table de fenêtre puis ouvre la fenêtre avec OPENW.

WINDTAB (de même que INTIN()) peut également être appelé en tant que tableau WINDTAB() à deux dimensions. Le premier index est le numéro de la fenêtre (1 à 4 ou 0), le second index étant :

0 pour le handle, 1 pour les attributs, 2 pour la coordonnée x, 3 pour la coordonnée y, 4 pour la largeur et 5 pour la hauteur de la fenêtre (dimension extérieure).

WINDTAB peut également être appelée à travers DPOKE, les différents offsets revêtant la signification suivante :

Offset	Fonction
0	Code (Handle) de la fenêtre 1
2	Table d'attributs de la fenêtre 1 (suivant la structure décrite plus loin)
4	Coordonnée gauche X de la fenêtre 1
6	Coordonnée supérieure Y de la fenêtre 1
8	Largeur de la fenêtre 1
10	Hauteur de la fenêtre 1
12 à 22	Les valeurs correspondantes pour la fenêtre 2.
24 à 34	Les valeurs correspondantes pour la fenêtre 3.
36 à 46	Les valeurs correspondantes pour la fenêtre 4.
48	-1
50	0
52 à 58	Coordonnées et dimensions pour la fenêtre du bureau (0).
60 à 62	Coordonnées du point de contact des fenêtres.
64 et 65	Point zéro pour les instructions graphiques (CLIP OFFSET).

Le point zéro n'est pas pris en compte par tous les appels AES, LINE-A ni par les appels VDI directs, ni par PUT, GET et BITBLT.

Cette table de 6 octets vaut pour la fenêtre 1. Des tables semblables existent pour les fenêtres 2, 3 et 4. Elles figurent dans les emplacements suivants de WINDTAB :

```
Fenêtre 2 : de WINDTAB(6)    à WINDTAB(11)
Fenêtre 3 : de WINDTAB+(12)  à WINDTAB(17)
Fenêtre 4 : de WINDTAB+(18)  à WINDTAB(23)
```

L'entrée d'un attribut de la fenêtre est organisée bit par bit, chaque bit correspondant à un élément de la fenêtre. Si un bit est mis, cela signifie que la fenêtre possède l'élément de fenêtre correspondant :

Bit	Elément de fenêtre correspondant
0	Entrée du titre de la fenêtre
1	Champ de fermeture en haut à gauche
2	Champ plein en haut à droite
3	Ligne de tête permettant de déplacer la fenêtre
4	Lignes d'informations sous la ligne titre
5	Champ de modification de la taille en bas à droite
6	Flèche vers le haut
7	Flèche vers le bas
8	Bouton-poussoir vertical à droite
9	Flèche vers la gauche
10	Flèche vers la droite
11	Bouton-poussoir horizontal en bas

Exemple :

```
' Il est également possible de simuler l'instruction
' OPENW #n,x,y,w,h,attr à travers WINDTAB, ce qui était la seule
' possibilité sous la version 2.0
'
OPENW #1,100,120,200,70,&HFFF
'
' équivaut à
'
DPOKE WINDTAB+2,&HFFF
DPOKE WINDTAB+4,100
DPOKE WINDTAB+6,120
DPOKE WINDTAB+8,200
DPOKE WINDTAB+10,70
OPENW 1
```

```
'
' ou
'
WINDTAB(1,1)=&HFFF
WINDTAB(1,2)=100
WINDTAB(1,3)=120
WINDTAB(1,4)=200
WINDTAB(1,5)=70
OPENW 1
```

DIVERS

RC_INTERSECT(x1,y1,w1,h1,x2,y2,w2,h2)

```
x1,y1,w1,h1 : iexp
x2,y2,w2,h2 : ivar
```

La fonction RC_INTERSECT (rectangle intersection) sert à déterminer si deux rectangles (définis par les coordonnées (x,y) du coin supérieur gauche ainsi que par la largeur w et la hauteur h) se chevauchent.

Si les deux rectangles se chevauchent, TRUE sera renvoyé et x2,y2,w2,h2 contiendront après appel de la fonction les coordonnées de la surface d'intersection. Si par contre il n'y a pas de surface d'intersection, FALSE sera renvoyé et x2,y2,w2,h2 contiendront les coordonnées d'un rectangle situé entre les deux rectangles spécifiés. La largeur w2 ou la hauteur h2 seront alors négatives ou nulles.

Cette fonction est normalement utilisée pour traiter les *'Redraws'* dans les événements GEM.

Exemple :

```
BOX 100,100,400,300
x=200
y=200
w=300
h=150
BOX x,y,x+w,y+h
'
IF RC_INTERSECT(100,100,300,200,x,y,w,h)
  PBOX x,y,x+w,y+h
ENDIF
~INP(2)
```

--> Deux rectangles sont dessinés et la surface d'intersection est représentée en noir.

RC_COPY s_adr,sx,sy,w,h **TO** d_adr,dx,dy [,m]

```
s_adr,d_adr,sx,sy,w,h,dx,dy,m : iexp
```

L'instruction RC_COPY permet de copier des rectangles d'image sur plusieurs pages d'image. Les paramètres *s_adr* et *d_adr* contiennent l'adresse des écrans source et de destination. Les coordonnées du coin supérieur gauche ainsi que la largeur et la hauteur du rectangle à copier sans transmises dans sx, sy, w et h. Les coordonnées du coin supérieur gauche du rectangle de destination sont dx et dy. Un mode de combinaison (0 à 15, cf. PUT) peut être spécifié en option. C'est la valeur 3 qui est prédéfinie pour m.

Exemple :

```
FILESELECT "\*.*","",f$
IF EXIST(f$)
  OPEN "i",#1,f$
  bild$=INPUT$(32000,#1)
  s_adr%=V:bild$
  d_adr%=XBIOS(2)
  '
  FOR i%=1 TO 1000
    RC_COPY s_adr%,RAND(10)*64,RAND(10)*40,64,40 TO d_adr%,RAND(10)*64,RAND(10)*40
  NEXT i%
  '
  CLOSE #1
ENDIF
```

--> Un fichier en format d'écran (32000 octets) est chargé puis affiché sur l'écran par sections découpées au hasard.

ALERT sym,texte$,default,button$,choix

```
(sym,default : iexp)
(texte$,button$ : sexp)
(choix : avar)
```

L'instruction ALERT fait apparaître un sélecteur d'alerte sur l'écran. L'expression *sym* définit quel symbole doit figurer dans la moitié gauche du sélecteur d'alerte, d'après les conventions suivantes :

```
0  -->  pas de symbole
1  -->  point d'exclamation
```

```
2  -->  point d'interrogation
3  -->  panneau Stop
```

L'expression de chaîne *texte$* contient le texte qui doit apparaître dans le sélecteur d'alerte. Ce texte peut comprendre jusqu'à 4 lignes de 30 caractères chacune au maximum. Les lignes sont séparées entre elles par un |. Les lignes de plus de 30 caractères sont tronquées.

L'expression *default* indique le numéro du bouton de sélection du sélecteur d'alerte qui représente ce qu'on appelle le bouton *défaut*. Ce bouton de sélection peut être sélectionné non seulement avec la souris mais aussi en appuyant sur la touche *Return* ou *Enter*. S'il n'existe aucun bouton portant le numéro *default*, cela entraîne que le sélecteur d'alerte ne peut être abandonné avec *Return* ou *Enter* mais seulement avec la souris.

L'expression de chaîne *button$* contient le texte du bouton du menu. La longueur de texte maximale par bouton est de 8 caractères. Les textes des différents boutons doivent être séparés entre eux par |.

Le numéro du bouton sélectionné est renvoyé dans la variable choix (cf. FORM_ALERT).

Exemple :

```
ALERT 1,"Choisissez|un bouton !",1,"Gauche|Droite",a%
ALERT 0,"Vous avez choisi le bouton "+STR$(a%)+".",0," Ok ",a%
```

--> Un sélecteur d'alerte avec deux boutons apparaît. Après qu'un bouton ait été sélectionné, un second sélecteur apparaît qui indique quel bouton a été sélectionné dans le premier sélecteur. Le second sélecteur d'alerte ne possède pas de symbole ni de bouton défaut.

FILESELECT chemin$,default$,nom$

```
chemin$,default$ : sexp
nom$ : svar
```

Cette instruction fait apparaître sur l'écran un sélecteur d'objet à l'aide duquel un fichier peut être sélectionné à l'intérieur du répertoire d'une disquette (ou d'un disque RAM, d'un disque dur, etc...).

L'expression *chemin$* contient le nom du lecteur et du dossier dont le répertoire doit être affiché. Si aucun lecteur n'est spécifié, c'est le lecteur actuel qui est sélectionné. *default$* est le nom du fichier qui doit être présélectionné à gauche dans le sélecteur de fichier.

nom$ contient le nom du fichier sélectionné une fois que le bouton Ok du sélecteur de fichier a été cliqué. Si le bouton *Arrêt* a été sélectionné, *nom$* contient une chaîne vide.

Le format de *chemin$, default$* et *nom$* est soumis aux conventions du système hiérarchisé de fichiers tel qu'il a été décrit dans la section **"Gestion de fichier"** du chapitre **"Entrées et sorties générales"**.

(cf. FSEL_INPUT)

Exemple :

```
FILESELECT "A:\*.PRG","GFABASIC.PRG",nom$
IF nom$=""
  PRINT "Vous avez sélectionné le bouton Arrêt."
ELSE IF RIGHT$(nom$)="\"
  PRINT "Vous avez sélectionné le bouton OK sans spécifier de fichier."
ELSE
  PRINT "Vous avez sélectionné le fichier : ";nom$;"."
ENDIF
```

--> Fait apparaître un sélecteur de fichier et analyse la sélection opérée par l'utilisateur.

10. ROUTINES SYSTEME

GEMDOS, BIOS et XBIOS

GEMDOS(no [,x,y...])
BIOS(no [,x,y...])
XBIOS(no [,x,y...])

```
(no,x,y : iexp)
```

Ces trois fonctions permettent d'appeler des routines GEMDOS, BIOS et XBIOS. A cet effet doivent être transmis le numéro de fonction no et une liste de paramètres. Pour que les paramètres puissent être transmis sous le format de variable qui convient, un W: ou un L: peuvent être placés devant les paramètres. Les paramètres désignés par un W: ainsi que les paramètres sans désignation seront alors transmis sous forme de mots (16 bits), les valeurs marquées par un L: étant transmises comme longs mots (32 bits). Voyez aussi l'annexe à se sujet.

Exemples :

```
IF GEMDOS(17)
  PRINT "Imprimante prête"
ELSE
  PRINT "Imprimante n'est pas prête"
ENDIF
```

--> Examine si une imprimante est prête sur l'interface parallèle (GEMDOS(17) renvoie l'état de sortie de l'interface parallèle).

```
REPEAT
UNTIL BIOS(11,-1) AND 4
```

--> Attend que la touche *Control* soit actionnée (BIOS(11,-1) renvoie l'état des touches de commutation du clavier).

```
CIRCLE 320,200,180
BMOVE XBIOS(2),XBIOS(2)+16000,16000
```

--> Dessine un cercle et en copie une moitié dans la moitié supérieure de l'écran (XBIOS(2) renvoie l'adresse à laquelle commence la mémoire physique de l'écran).

L:x
W:x

```
x : iexp
```

Ces deux fonctions servent à transmettre des expressions numériques sous forme de mots (2 octets, W:) ou de longs mots (4 octets, L:) lors de l'appel de fonctions du système d'exploitation et de routines C. Si la marque W: ou L: fait défaut, c'est le format de mot qui est automatiquement utilisé.

Exemple :

```
CLS
FOR i% = 1 TO 100
  BOX RAND(639),RAND(399),RAND(639),RAND(399)
NEXT i%
PBOX 0,0,639,399
DIM screen_2|(32255)
base_phys%=XBIOS(2)
ancien_ecran%=base_phys%
base_log%=V:screen_2|(0)+255 AND &HFFFFFF00
REPEAT
  IF MOUSEK=1
     ~XBIOS(5,L:base_log%,L:base_phys%,-1)
     SWAP base_log%,base_phys%
  ENDIF
  PLOT MOUSEX,MOUSEY
UNTIL MOUSEK=2
~XBIOS(5,L:ancien_ecran%,L:ancien_ecran%,-1)
```

--> Commute entre deux pages écran lorsque vous appuyez sur le bouton gauche de la souris. Le fait d'appuyer sur le bouton droit de la souris entraîne l'arrêt du programme (voyez aussi XBIOS). La position de la souris est en outre plottée sur les deux pages d'image.

Le nombre et le rôle des paramètres ainsi que la valeur de réponse renvoyée dépendent de la routine du système d'exploitation qui a été appelée.

APPELS LINE-A

ACLIP, PSET, PTST, ALINE, HLINE, ARECT, APOLY, ACHAR, ATEXT

La présente section décrit des instructions qui équivalent à d'autres instructions déjà présentées dans le chapitre consacré au graphisme. Les sorties graphiques à travers les appels LINE-A sont toutefois nettement plus rapides et emploient une syntaxe quelque peu différente. Avant des sorties graphiques LINE-A, il est recommandé de fixer systématiquement des limites (clipping) avec ACLIP car les zones de mémoire situées en dehors de la mémoire écran seront sinon effacées. On emploie souvent aussi le clipping de n'importe quelle instruction graphique (VDI ou AES) antérieure.

Un appel VDI modifie le clipping mis en place avec ACLIP. Les appels LINE-A sont indépendants des instructions VDI DEFxxx.

En ce qui concerne la sélection des couleurs : les couleurs indiquées pour les routines Line-A correspondent aux numéros de registres de couleur électroniques (comme avec SETCOLOR) et non aux numéros employés par la VDI (COLOR).

ACLIP flag,xmin,ymin,xmax,ymax

```
flag,xmin,ymin,xmax,ymax : iexp
```

Cette instruction permet de définir un rectangle de délimitation, c'est-à-dire que toutes les sorties graphiques avec LINE-A seront limitées à cette section rectangulaire de l'écran. xmin,ymin, xmax et ymax contiennent les coordonnées des coins supérieur gauche et inférieur droit du rectangle. Avec flag<>0, la délimitation est activée, sinon elle est désactivée. ACLIP ne vaut (malheureusement) pas pour PSET, PTST, ALINE, HLINE et BITBLT.

PSET x,y,c

```
(x,y,c : iexp)
```

PSET équivaut à l'instruction PLOT et permet de fixer des points de couleur 'c' sur les coordonnées x,y. 'c' peut revêtir une valeur de 0 à 15, en fonction de la résolution.

Exemple :

```
FOR i%=0 TO 199 STEP 2
  PSET i%,i%,1
NEXT i%
```

--> Fixe des points espacés de deux points écran de 0,0 à 199,199.

PTST(x,y)

```
(x,y : iexp)
```

La fonction PTST équivaut à la fonction POINT. Elle renvoie la couleur de la position x,y de l'écran.

Exemple :

```
PSET 100,100,1
x=PTST(200,100)
PRINT x,PTST(100,100)
```

--> Affiche le code des couleurs des positions 200,100 et 100,100 de l'écran.

ALINE x1,y1,x2,y2,c,ml,m

```
(x1,y1,x2,y2,c,ml,m : iexp)
```

ALINE équivaut à l'instruction LINE. x1,y1,x2,y2 contiennent ici les coordonnées des points de départ et de fin de la ligne. L'expression 'c' contient les informations de couleur. Elle revêt une valeur de 0 à 15 en fonction de la résolution (voyez COLOR). 'ml' contient les informations de bits définissant le motif de ligne voulu. Il s'agit d'un masque d'une longueur de 16 bits dont chaque bit mis (en mode monochrome) correspond à un point à dessiner.

Le paramètre 'm' fixe le mode graphique. Il peut revêtir des valeurs de 0 à 3, d'après les conventions suivantes :

```
0 --> Remplacer
1 --> Transparent
2 --> Inverser
3 --> Inverser transparent
```

Exemple :

```
ym%=INT(L-A-4)-1
FOR i%=0 TO 199
  ALINE i%,0,i%,ym%,1,&HFFFF,0
NEXT i%
```

--> Dessine des lignes verticales sur toute la hauteur de l'écran de la coordonnée x 0 à la coordonnée x 199.

HLINE x1,y,x2,c,m,adr,nmb_motifs

```
x1,x2,y,c,m,adr,nmb_motifs : iexp
```

HLINE équivaut à l'instruction LINE mais cette instruction ne permet de dessiner que des lignes horizontales. x1 et x2 contiennent ici les deux coordonnées x et y la coordonnée y de la ligne à dessiner. L'expression 'c' contient les informations de couleur. Elle peut revêtir des valeurs de 0 à 15 en fonction de la résolution (voyez COLOR). Le paramètre 'm' fixe le mode graphique et correspond à celui de ALINE.

'adr' contient l'adresse d'un tableau dans lequel figurent les informations de bits pour les motifs de ligne (16 bits par motif). L'expression *'nmb_motifs'* est un masque qui sera combiné avec la coordonnée Y pour fournir un index pour la table des motifs de ligne. C'est pourquoi *nmb_motifs* devra en principe être inférieur d'une unité à une puissance de 2 (0,1,3,7,15,...).

Exemple :

```
ACLIP 1,0,0,639,399
'
motif%=&X1111111111111111101010101010101O
z%=V:motif%
'
FOR i%=0 TO 199
  HLINE 0,i%,639,1,0,z%,1
NEXT i%
```

--> Deux motifs de ligne de 16 bits sont stockés dans la variable *motif%*. Le dernier paramètre de HLINE est donc 1 (2 motifs). Les lignes dessinées avec HLINE utilisent ensuite alternativement les deux motifs de ligne de 16 bits figurant dans *motif%*.

ARECT x1,y1,x2,y2,c,m,adr,nmb_motifs

```
x1,y1,x2,y2,c,m,adr,nmb_motifs : iexp
```

ARECT équivaut à PBOX. x1,y1 et x2,y2 contiennent ici les coordonnées de deux coins opposés du rectangle à dessiner. Les paramètres *c, m, adr* et *nmb_motifs* ont la même signification que pour HLINE.

Exemple :

```
ACLIP 1,0,0,639,399
DIM motif%(1)
'
motif%(0)=&X1010101010101010
motif%(1)=&X0101010101010101
'
adr_motif%=V:motif%(0)
ARECT 100,100,200,200,1,0,adr_motif%,1
```

--> Dessine un rectangle rempli avec un motif semblable au fond du bureau (c'est-à-dire DEFFILL 1,2,4).

APOLY adr_pnt,nmb_pnt,y0 **TO** y1,c,m,adr,nmb_motifs

```
adr_pnt,nmb_pnt,y0,y1,c,m,adr,nmb_motifs : iexp
```

APOLY ressemble à l'instruction POLYFILL et dessine un tracé invisible de ligne brisée fermée de n angles qui est rempli avec un motif quelconque. On transmet à cet effet dans *adr_pnt* l'adresse du tableau qui contient alternativement les coordonnées x et y des points. Le paramètre *'nmb_pnt'* contient le nombre de points. y0 et y1 indiquent les coordonnées Y définissant la partie du polygone qui devra être remplie. Les paramètres *c, m, adr* et *nmb_motifs* ont la même signification que pour HLINE.

Exemple :

```
DIM x&(9),y&(9),motif&(1)
FOR i%=0 TO 8
  x&(i%)=RAND(100)
NEXT i%
x&(9)=x&(0)
'
adr_cor%=V:x&(0)
motif&(0)=-1
```

```
adr_motif%=V:motif&(0)
'
ACLIP 1,0,0,200,200
'
FOR i%=1 TO 8
  APOLY adr_xcor%,9,0 TO 100,1,1,adr_motif%,0
NEXT i%
```

--> Dessine un polygone fermé qui est ensuite rempli.

BITBLT adr%
BITBLT x%()

```
adr% : iexp
x%() : tableau entier de quatre octets
```

L'instruction BITBLT appelle la routine Line-A homonyme. BITBLT appelle une routine VDI avec trois tableaux de paramètres. Avec la variante de l'instruction pour laquelle une adresse doit être spécifiée comme paramètre, une table de 76 octets doit figurer à partir de cette adresse. La table suivante indique la signification des différentes valeurs que renferme cette zone de mémoire, la colonne offset indiquant la distance en octets de chaque élément de cette table par rapport au début de la table.

Avec la variante de l'instruction pour laquelle un tableau d'entiers de quatre octets d'au moins 23 éléments doit être spécifié, un paramètre figure dans chaque élément du tableau. La colonne Index de la même table indique dans quel élément du tableau doit figurer chaque paramètre.

Les paramètres marqués d'une étoile sont modifiés par la routine BITBLT. C'est pourquoi il est normalement conseillé de travailler avec la variante de l'instruction employant un tableau comme paramètre car une copie des paramètres y figurant est produite dans ce cas et c'est à cette copie que la routine BITBLT accède, les éléments du tableau n'étant donc pas modifiés. Les éléments du tableau seront au contraire modifiés lorsqu'une adresse est transmise.

Nom	Index	Offset	Signification
B_WD	00	00	Largeur de grille en points écran.
B_HT	01	02	Hauteur de grille en points écran.
PLANE_CT *	02	04	Nombre de plans de couleurs.
FG_COL *	03	06	Couleur de premier plan
BG_COL *	04	08	Couleur du fond
OP_TAB	05	10	Combinaison logique pour toute combinaison entre les points écran de premier plan et du fond.
S_XMIN	06	14	Offset x dans la grille source
S_YMIN	07	16	Offset y dans la grille source
S_FORM	08	18	Adresse de la grille source
S_NXWD	09	22	Offset jusqu'au prochain mot sur le même niveau.
S_NXLN	10	24	Offset jusqu'à la prochaine ligne de la grille source
S_NXPL	11	26	Offset jusqu'au prochain niveau de couleur (2).
D_XMIN	12	28	Offset x dans la grille de destination.
D_YMIN	13	30	Offset y dans la grille de destination.
D_FORM	14	32	Adresse de la grille de destination.
D_XNWD	15	36	Offset jusqu'au prochain mot du même niveau.
D_NXLN	16	38	Offset jusqu'à la prochaine ligne de la grille de destination.
D_NXPL	17	40	Offset jusqu'au prochain niveau de couleur (toujours 2).
P_ADDR	18	42	Pointeur sur la table avec motif de remplissage ; si différent de zéro, combinaison avec AND.
P_NXLN	19	46	Offset jusqu'à la prochaine ligne de la grille de masque.
P_NXPL	20	48	Offset jusqu'à la prochaine couleur du motif de remplissage.
P_MASK	21	50	Masque comme pour HLINE,
SPACE *	22	52	Les 24 octets suivants servant de zone de travail pour le blitter.

Exemples :

```
DIM x%(1000)
'
FOR i%=0 TO 639 STEP 8
  CIRCLE i%,i%,399
NEXT i%
'
GET 0,0,639,399,a$
```

```
mirrorput(0,0,a$)
'
PROCEDURE mirrorput(x%,y%,VAR x$)
  IF LEN(x$)>6      !Seulement s'il s'y trouve bien quelque chose
    xx%=V:x%(0)
    a%=V:x$
    b%=INT{a%}
    h%=INT{a%+2}
    '
    INT{xx%}=1
    INT{xx%+2}=h%
    INT{xx%+4}=1
    INT{xx%+6}=1
    INT{xx%+8}=0
    {xx%+10}=&H3030303
    INT{xx%+14}=9999
    INT{xx%+16}=0
    {xx%+18}=a%+6
    INT{xx%+22}=2
    INT{xx%+24}=SHR(b%+16,4)*2
    INT{xx%+26}=2
    INT{xx%+28}=9999
    INT{xx%+30}=0
    {xx%+32}=XBIOS(3)
    INT{xx%+36}=2
    INT{xx%+38}=80
    INT{xx%+40}=2
    {xx%+42}=0    !pattadr
    INT{xx%+46}=0    !p_nxtln
    INT{xx%+48}=0    !p_nxpl
    INT{xx%+50}=0    !p_mask
    '
    ABSOLUTE i&,xx%+14
    ABSOLUTE di&,xx%+28
    '
    FOR i&=0 TO b%
      INT{xx%+4}=1
      di&=SUB(639,i&)
      BITBLT xx%
    NEXT i&
    '
  ENDIF
RETURN
```

--> Exemple avec BITBLT adr%.

```
DIM x%(1000)
'
FOR i%=0 TO 639 STEP 8
  CIRCLE i%,i%,399
NEXT i%
'
GET 0,0,639,399,a$
mirrorput(0,0,a$)
'
PROCEDURE mirrorput(x%,y%,VAR x$)
  IF LEN(x$)>6      !Seulement s'il s'y trouve bien quelque chose
    a%=V:x$
    b%=INT(a%)
    h%=INT(a%+2)
    '
    x%(0)=1
    x%(1)=h%
    x%(2)=1
    x%(3)=1
    x%(4)=0
    x%(5)=&H3030303
    x%(6)=9999
    x%(7)=0
    x%(8)=V:x$+6
    x%(9)=2
    x%(10)=SHR(b%+16,4)*2
    x%(11)=2
    x%(12)=9999
    x%(13)=0
    x%(14)=XBIOS(3)
    x%(15)=2
    x%(16)=80
    x%(17)=2
    x%(18)=0    !pattadr
    x%(19)=0    !p_nxtln
    x%(20)=0    !p_nxpl
    x%(21)=0    !p_mask
    '
    FOR i%=0 TO b%
      x%(6)=i%
      x%(12)=639-i%
      BITBLT x%()
    NEXT i%
    '
  ENDIF
RETURN
```

--> Exemple avec BITBLT x%().

Ces deux routines reflètent sur l'écran une section lue avec GET, comme le listing avec BITBLT dans la chapitre **Instructions graphiques générales.** Dans la routine employant BITBLT *adr%*, le nombre de plans de bits doit être chaque fois fixé à nouveau. Le choix de la routine BITBLT à employer est surtout une question de préférence personnelle car il n'y a pas de grande différence du point de vue de la vitesse d'exécution.

ACHAR code,x,y,fonte,style,angle

```
(code,x,y,fonte,style,angle : iexp)
```

ACHAR permet de sortir un caractère isolé correspondant à n'importe quel code ASCII 'code' sur le point (x,y). L'expression numérique *fonte* peut ici revêtir des valeurs entre 0 et 2. Les jeux de caractères suivants sont attribués à ces valeurs :

```
0 = 6x6  (Ecriture d'icône)
1 = 8x8  (Ecriture normale en mode couleur)
2 = 8x16 (Ecriture monochrome normale)
```

Les valeurs plus élevées pour *fonte* seront interprétées comme des adresses de *header* de fonte. Une fonte de ce type doit se présenter sous le format dans lequel les fontes sont converties lorsqu'elles sont chargées par le GDOS, c'est-à-dire en format Motorola avec l'octet fort avant l'octet faible.

Le style du texte (0 à 31) et l'angle de sortie (0,900,1800,2700) peuvent également être définis. Ces valeurs ont la même signification que celles employées avec l'instruction TEXT. Toutefois, contrairement à ce qui est le cas avec l'instruction TEXT, les coordonnées spécifiées se rapportent au bord supérieur du caractère.

ATEXT x,y,fonte,s$

```
x,y,fonte : iexp
s$ : sexp
```

L'instruction ATEXT sert à sortir des chaînes de caractères dans n'importe quelles positions de l'écran. Contrairement à ce qui est le cas avec ACHAR, il n'est cependant pas possible de définir un style d'écriture ou un angle de sortie déterminé. Les paramètres x,y et fonte ont la même signification que pour ACHAR.

Exemple :

```
a_clock                          ! appelle la procédure a_clock
'                                  une première fois
EVERY 400 GOSUB a_clock          ! Appeler a_clock toutes les
'                                  2 secondes
OPENW 0                          ! Protéger la ligne du haut contre
'                                ! tout empiètement
FOR i%=1 TO 100000
  PRINT USING " ###### ",i%;     ! sinon sortir des chiffres
NEXT i%
'
PROCEDURE a_clock
  ACLIP 1,20,0,120,15            ! Activer clipping, sinon c'est le
  '                                clipping de OPENW 0 qui
  '                                s'applique, autrement dit : on ne
  '                                voit rien.
  ATEXT 20,0,2,TIME$             ! Afficher l'heure
RETURN
```

--> Affiche l'heure toutes les 2 secondes et écrit des colonnes de nombres sur l'écran.

L~A

La fonction L~A détermine l'adresse de base des variables Line-A (voyez l'annexe).

Exemple :

```
PRINT INT(L~A)
```

--> Sort le nombre de plans de bits sous la résolution actuelle. Voyez aussi les exemples donnés sous BITBLT.

APPELS VDI

Les fonctions VDI se divisent en 7 domaines :

- *Fonctions de contrôle*
- *Fonctions de sortie*
- *Fonctions d'attribut*

- *Fonctions de grille*
- *Fonctions d'entrée*
- *Fonctions de renseignements*
- *Escapes*

On distingue 3 sortes de paramètres : entrée, sortie et les paramètres employés aussi bien pour l'entrée que pour la sortie. Ces paramètres sont transmis à travers 5 tableaux :

- *CONTRL*	Contrôle
- *INTIN*	Entrée entière
- *PTSIN*	Entrée de coordonnées de point
- *INTOUT*	Sortie entière
- *PTSOUT*	Sortie de coordonnées de point

Les paramètres en entrée figurent dans :

CONTRL(0)	Code d'opération
CONTRL(1)	Nombre de points dans le tableau PTSIN
CONTRL(3)	Longueur du tableau INTIN
CONTRL(5)	Identification du code d'opération
INTIN()	Tableau d'entrée des valeurs entières
PTSIN()	Tableau d'entrée des coordonnées de point

Les paramètres en sortie sont :

CONTRL(2)	Nombre de points dans le tableau PTSOUT
CONTRL(4)	Longueur du tableau INTOUT
INTOUT()	Tableau de sortie des valeurs entières
PTSOUT()	Tableau de sortie des coordonnées de point

Les paramètres en entrée et sortie sont :

CONTRL(6)	Code de périphérique
CONTRL(7-n)	Informations liées au code d'opération

Les valeurs qui doivent être inscrites dans les différents champs lors de l'appel d'une fonction VDI varient d'une fonction à l'autre.

CONTRL
INTIN
PTSIN
INTOUT
PTSOUT

Ces fonctions déterminent les adresses des blocs de paramètres VDI. Un index entre parenthèses permet d'appeler les éléments de ces tableaux. Les adresses sont :

```
CONTRL --> Adresse du tableau de contrôle VDI
INTIN  --> Adresse du tableau d'entrée entière VDI
PTSIN  --> Adresse du tableau d'entrée des coordonnées de point VDI
INTOUT --> Adresse du tableau de sortie entière VDI
PTSOUT --> Adresse du tableau de sortie de coordonnées de point VDI
```

Ces tableaux contiennent les paramètres pour les appels VDI.

CONTRL(2) équivaut par exemple à DPOKE CONTRL+4 ou à DPEEK(CONTRL+4). Les autres tableaux sont également organisés mot par mot.

VDISYS [opc [,c_int,c_pts [,subopc]]]]

```
opc,c_int,c_pts,subopc : iexp
```

L'instruction VDISYS appelle la fonction VDI numéro *opc*. Si *opc* n'est pas spécifié, le numéro de fonction ainsi que les autres paramètres doivent être inscrits avec DPOKE CONTRL,opc ou CONTRL(0)=opc dans le tableau de contrôle.

Dans les paramètres *c_int* et *c_pts* peut être indiqué le nombre de valeurs dans le tableau d'entrée entière et dans le tableau d'entrée de coordonnées de point. Il devient alors inutile d'inscrire ces valeurs dans le tableau de contrôle. Le paramètre optionnel *subopc* contient le sous-code d'opération de la routine VDI à appeler. Ce sous-code ne doit être indiqué que pour quelques routines VDI telles que les routines *Escape* par exemple.

Les paramètres *c_int*, *c_pts* et *subopc* sont donc inscrits dans des entrées déterminées du tableau CONTRL.

Voici les conventions utilisées :

```
opc    --> CONTRL(0)
c_int  --> CONTRL(3)
c_pts  --> CONTRL(1)
subopc --> CONTRL(5)
```

Exemples :

```
CONTRL(1)=3
CONTRL(5)=4
PTSIN(0)=320
PTSIN(1)=200
PTSIN(4)=190
VDISYS 11

PTSIN(0)=320
PTSIN(1)=200
PTSIN(4)=190
VDISYS 11,0,3,4

PCIRCLE 320,200,190
```

--> Dessine trois fois un cercle plein sur l'écran.

```
VDISYS 5,0,0,13
PRINT " Inversé "
VDISYS 5,0,0,14
PRINT " Normal "
```

--> Ecrit le texte *'Inversé'* inversé puis le texte *'Normal'* non inversé.

VDIBASE

La variable système VDIBASE contient l'adresse à partir de laquelle la version actuelle de GEM stocke les paramètres chargés de la gestion des routines VDI. L'organisation de cette zone de la mémoire peut changer dans les versions ultérieures de GEM. C'est pourquoi nous ne vous fournirons pas d'exemple pour ne pas vous encourager à user de cette possibilité. VDIBASE existe sous GFA-BASIC uniquement pour donner aux programmeurs la possibilité d'accéder à tous les paramètres VDI.

WORK_OUT(x)

```
x : iexp
```

Cette fonction détermine les valeurs figurant dans intout(0) à intout(44) et dans ptsout(0) à ptsout(1) lors de l'appel de la fonction VDI OPEN WORKSTATION. L'indice va cependant de 0 à 56 lorsque les fonctions WORK_OUT sont employées (voyez l'annexe).

Exemple :

```
largeur_grille&=WORK_OUT(0)
hauteur_grille&=WORK_OUT(1)
PRINT largeur_grille&,hauteur_grille&,WORK_OUT(10)
```

--> Renvoie les nombres 639 et 399 pour les largeur et hauteur de grille (en mode monochrome) et un 1 pour le nombre de jeux de caractères disponibles (WORK_OUT(10)).

Routines VDI spéciales et GDOS

Les fonctions de station de travail et de renseignement VDI suivantes ne sont disponibles que si le programme GDOS (à partir de la version 1.0) a été chargé lors du lancement du système et si un fichier ASSIGN.SYS valable est présent.

Le GDOS (Graphics Device Operating System), indépendant de la machine, contient des fonctions graphiques et collabore avec des drivers indépendants de la machine pour différents périphériques de sortie (écran, imprimante, plotter, metafile, etc...).

Le fichier ASCII ASSIGN.SYS contient toutes les indications nécessaires sur la configuration de la machine utilisée. Dans ce fichier doivent être inscrits tous les drivers de périphériques et tous les jeux de caractères utilisés. Le chemin d'accès des drivers de périphériques doit être spécifié le cas échéant lorsque ceux-ci ne figurent pas sur le lecteur A:. La syntaxe suivante doit être respectée pour ce faire :

```
id  DRIVER.SYS;
;   Commentaires
    jeu1.FNT
    jeu2.FNT
    ...
    jeun.FNT
```

id contient un nombre de deux chiffres entre 1 et 32767 qui fixe le type du driver de périphérique, d'après les définitions suivantes :

```
01 ...   Ecran
11 ...   Plotter
21 ...   Imprimante
31 ...   Fichiers Meta
41 ...   Caméra
51 ...   Tableur graphique
```

Si les drivers de périphériques figurent sur le lecteur C: dans le dossier GEMSYS, le fichier ASSIGN.SYS pourra alors se présenter par exemple ainsi :

```
path = c:\gemsys\
01p screen.sys;       Driver d'écran de la ROM, d'où le p après l'id
ATSS10.FNT
ATSS12.FNT
02p screen.sys;       Driver pour la basse résolution
ATSS10.FNT
ATSS12.FNT
03p screen.sys;       Driver pour la résolution moyenne
ATSS10CG.FNT
ATSS12CG.FNT
04p screen.sys;       Driver pour la haute résolution
ATSS10.FNT
ATSS12.FNT
21 fx80.sys;          Driver d'imprimante pour la FX-80 et compatible
ATSS10EP.FNT
ATSS12EP.FNT
31 META.SYS;          Driver de metafile
ATSS10MF.FNT
ATSS12MF.FNT
```

Les drivers d'écran SCREEN.SYS sont simplement des entrées fictives qu'impose le contrôle de syntaxe opéré par GDOS. Les id (2, 3 et 4) servent uniquement à affecter les fontes appropriées à chaque résolution. C'est pourquoi pour ouvrir une station de travail d'écran virtuelle l'appel se présentera ainsi : V_OPNVWK(XBIOS(4)+2). Cet appel est effectué par GFA-BASIC 3.0 en interne.

GDOS?

GDOS? renvoie TRUE si GDOS (à partir de la version 1.0) est chargé et FALSE si GDOS n'est pas résident.

Exemple :

```
IF NOT GDOS?
  ALERT 1,"GDOS non trouvé !",1,"Arrêt",r%
  END
ENDIF
```

V~H

Dans les fonctions de contrôle VDI suivantes, qui sont adaptées sous GFA-BASIC, hd représentera le code du programme et id le code du périphérique de sortie. Il convient à cet égard de noter les fonctions suivantes :

V~H fournit le handle VDI utilisé à l'intérieur de GFA-BASIC (par exemple PRINT V~H).

V~H=x fixe sur la valeur x le handle VDI utilisé sur le plan interne (auquel il peut être accédé avec CONTRL(6)).

V~H=-1 fixe sur la valeur fournie par V_OPNVWK (qui est utilisée en interne) le handle VDI utilisé sur le plan interne (auquel il peut être accédé avec CONTRL(6)).

V_OPNWK(id)
V_OPNWK(id,1,1,1,1,1,1,1,1,1,2)
V_CLSWK()

```
id : iexp
```

Les nombres 1 ou 2 affichent les valeurs défaut pour le réglage des paramètres VDI, ils peuvent également être modifiés (cf. WORK_OUT).

La fonction V_OPNWK (open work station) fournit le handle hd% correspondant au code de périphérique id% spécifié. D'autres informations sur le périphérique connecté peuvent être en outre obtenues à l'aide de INTOUT() et de PSTOUT() (cf. VDISYS dans la section consacrée aux routines système).

La fonction V_CLSWK (close work station) referme la station de travail ouverte (avec V_OPNWK) et écrit le contenu intégral du buffer de cette station de travail. Un V~H=-1 est en outre exécuté.

Exemple :

(cf. sous V_CLRWK).

V_OPNVWK(id)
V_OPNVWK(id,1,1,1,1,1,1,1,1,1,2)
V_CLSVWK(id)

```
id : iexp
```

La fonction V_OPNVWK (open virtual work station) ouvre un driver d'écran virtuel et fournit le handle pour le code de périphérique id spécifié (cf. V_OPNWK).

La fonction V_CLSVWK (close virtual work station) ferme une station de travail virtuelle ouverte (avec V_OPNVWK). Un V~H=-1 est en outre exécuté.

V_CLRWK()
V_UPDWK()

La fonction V_CLRWK (clear work station) vide le buffer de sortie. Cela entraîne par exemple un vidage de l'écran ou du buffer d'imprimante.

Lors de sorties graphiques sur l'imprimante, toutes les instructions sont rassemblées dans un buffer. La fonction V_UPDWK (update work station) sort sur le périphérique connecté les instructions graphiques ainsi stockées dans un buffer. Sur l'écran par contre, toutes les instructions graphiques sont immédiatement exécutées.

Exemple :

```
RESERVE 25600                   ! Réserver suffisamment de place
'                                 mémoire
handle%=V_OPNWK(21)             ! Déterminer le code du
'                                 périphérique de sortie
IF handle%=0
  ALERT 3,"Erreur d'installation !",1,"Arrêt",r%
  RESERVE
  END
ENDIF
'
x_res%=INTOUT(0)                ! Déterminer résolutions x et y
y_res%=INTOUT(1)                ! du périphérique connecté
'
V~H=handle%                     ! fixe le handle VDI interne sur
```

```
'                                    le code d'imprimante
~V_CLRWK()                          ! Vider buffer
'
CLIP 0,0,x_res%,y_res%

BOX 0,0,x_res%,y_res%
LINE 0,0,x_res%,y_res%
LINE 0,y_res%,x_res%,0
'
~V_UPDWK()                          ! Exécution instructions
'                                    graphiques
~V_CLSWK()
'
RESERVE                             ! et restituer la mémoire
```

--> Si GDOS est activé sort sur une imprimante connectée un rectangle avec diagonales dessinées.

VST_LOAD_FONTS(x)
VST_UNLOAD_FONTS(x)

```
x: iexp
```

La fonction VST_LOAD_FONTS charge les jeux de caractères supplémentaires spécifiés dans ASSIGN.SYS si la place mémoire disponible est suffisante, et renvoie le nombre d'écritures effectivement chargées. S'il n'y a plus d'autres fontes disponibles, zéro est renvoyé.

Il est conseillé de fixer le paramètre x sur zéro en prévision d'éventuelles extensions dans les versions futures de GEM. Il importe avant tout que suffisamment de place mémoire ait été réservée avec RESERVE pour les jeux de caractères supplémentaires.

La fonction VST_UNLOAD_FONTS élimine de la mémoire les jeux de caractères chargés préalablement avec VST_LOAD_FONTS. En ce qui concerne le paramètre, ce qui a été dit sous VST_LOAD_FONTS s'applique également ici.

Exemple :

```
RESERVE 25600
'
nmb_fontes%=VST_LOAD_FONTS(0)       ! Combien de jeux de caractères
                                    ! supplémentaires ?
face%=VQT_NAME(nmb_fontes%,fonte$) ! Index et nom de la fonte
'                                    chargée
```

```
FOR i%=1 TO nmb_fontes%
  DEFTEXT ,,,,face%
  TEXT 80,80,"Voici la fonte "+font$+"."
  ~INP(2)
NEXT i%
'
~VST_UNLOAD_FONTS(0)                ! Eliminer fontes
'
RESERVE
```

--> Sort sur l'écran le nom des jeux de caractères chargés sous les fontes correspondantes.

VQT_EXTENT(texte$ [,x1,y1,x2,y2,x3,y3,x4,y4])

```
texte$ : sexp
x1,y1,x2,y2,x3,y3,x4,y4: ivar
```

La fonction VQT_EXTENT fournit les dimensions d'un rectangle renfermant la chaîne de caractères texte$ spécifiée. La réponse peut s'effectuer au choix à travers les variables x1,y1 à x4,y4 ou seulement dans PTSOUT(0) à PTSOUT(7). Les coins sont numérotés dans le sens contraire des aiguilles d'une montre.

Coin	Position
x1,y1	Coin inférieur gauche.
x2,y2	Coin inférieur droit.
x3,y3	Coin supérieur droit.
x4,y4	Coin supérieur gauche.

VQT_NAME(i,nom_fonte$)

```
i : ivar
nom_fonte$: svar
```

La fonction VQT_NAME fournit le code du jeu de caractères chargé numéro i et inscrit le nom de cette fonte dans la variable de chaîne nom_fonte$.

Exemple :

```
RESERVE 2560
'
```

```
nmb_fontes%=VST_LOAD_FONTS(0)
face%=VQT_NAME(nmb_fontes%,fonte$) ! Index et nom d'une fonte
'                                    chargée
h%=12                              ! Hauteur de texte
s$="Texte d'exemple"
x0%=80                             ! Coordonnées de sortie
y0%=80                             ! pour s$
DEFTEXT 1,0,0,h%,face%
'
~VQT_EXTENT(s$,x1%,y1%,x2%,y2%,x3%,y3%,x4%,y4%)
'
GRAPHMODE 4
TEXT x0%,y0%,s$
PBOX x0%+x1%,x0%+y1%-h%-1,x0%+x3%,y0%+y3%-h%
'
~VST_UNLOAD_FONTS(0)               ! Eliminer fontes
'
RESERVE
```

--> La chaîne s$ est sortie inversée sur le point x0,y0 de l'écran.

APPEL DE SOUS-PROGRAMMES D'AUTRES LANGAGES

Les instructions décrites dans cette section servent à appeler des sous-programmes écrits en C ou en assembleur.

C:adr([x,y,...])

```
adr : avar (au moins 32 bits, adr% de préférence)
x,y : iexp
```

La fonction C appelle un sous-programme écrit en C ou en assembleur situé à l'adresse *adr*. Les paramètres x,y,... peuvent être transmis entre les parenthèses. La transmission de paramètres se déroule comme en C. Les paramètres peuvent être transmis sous forme de longs mots de 32 bits ou de mots de 16 bits, la transmission de valeurs 16 bits étant sélectionnée a priori. Les longs mots peuvent être transmis en plaçant un L: devant le paramètre correspondant. Lors de l'appel de cette fonction, l'adresse de retour puis les paramètres sont placés sur la pile. C'est ainsi par exemple que :

```
VOID C:adr%(L:x,W:y,z)
```

aboutira sur la pile à la situation suivante :

```
  (sp) --> Adresse de retour
 4(sp) --> x (4 octets)
 8(sp) --> y (2 octets)
10(sp) --> z (2 octets)
```

La valeur renvoyée par la fonction est le contenu du registre d0 lors du retour du sous-programme (retour qui doit s'effectuer avec un RTS).

Exemple :

Le programme assembleur ici utilisé remplit une zone de la mémoire (par exemple un tableau entier) à partir d'une adresse déterminée avec les nombres (longs mots) de 0 à n.

```
206F0004      move.l   4(sp),a0  ;Adresse de départ
202F0008      move.l   8(sp),d0  ;Nombre de valeurs
7200          moveq.l  #0,d1     ;Compteur
6004          bra.s    ct_2      ;Entrée dans boucle
20C1    ct_1:move.l    d1,(a0)+  ;Boucle, écrire valeur
5281          addq.l   #1,d1     ;Augmenter compteur
B081    ct_2:cmp.l     d1,d0     ;Terminé ?
64F8          bcc.s    ct_1      ;Non, suite -->
4E75          rts                ;Retour à GFA-BASIC
```

Le programme BASIC sera :

```
FOR i%=1 TO 11
  READ a%
  asm$=asm$+MKI$(a%)
NEXT i%
DATA $206F,$0004,$202F,$0008,$7200,$6004,$20C1,$5281,$B081,$64F8,$4E75
'
DIM x%(10000)
asm%=V:asm$
~C:asm%(L:V:x%(0),L:10000)
PRINT "Par ex. x%(12): ";x%(12)
```

--> Remplit le tableau x%() avec les nombres 0 à 10000. Dans cette application, l'exécution de la fonction C: équivaut aux instructions :

```
FOR i%=1 TO n%
  x%(i%)=i%
NEXT i%
```

L'instruction INLINE offre une autre possibilité d'intégration d'un programme assembleur sous GFA-BASIC 3.0.

Il faut pour cela créer tout d'abord un fichier contenant le programme assembleur imprimé ci-dessus. A cet effet, après la boucle READ-DATA indiquée plus haut, entrez :

```
BSAVE "COUNT.INL",V:asm$,22
```

Puis entrez le programme suivant :

```
INLINE asm%,22
DIM x%(10000)
~C:asm%(L:V:x%(0),L:10000)
```

Appuyez ensuite sur la touche *Help* dans la ligne où figure INLINE, puis sélectionnez *Load* et chargez COUNT.INL. Le programme peut alors être sauvegardé avec *Save*.

MONITOR [x]

```
x : iexp
```

Cette instruction sert à appeler des routines assembleur, des programmes de débogage ou d'autres programmes utilitaires. A cet effet, le vecteur *Illegal Instruction* (Adresse 16) doit être détourné sur l'adresse des sous-programmes voulus. L'instruction MONITOR produit alors une *Illegal Instruction Exception* qui entraîne un saut au sous-programme, qui doit se terminer par RTE (ReTurn from Exception). Le paramètre x est une valeur de communication qui sera écrite dans le registre d0.

Un exemple d'application consisterait à employer un débogueur dans un programme tel que celui décrit pour C:. A cet effet, il faut charger et lancer le GFA-BASIC à partir du débogueur. Ensuite, après avoir entré le programme, insérer : MONITOR *asm%* directement à la suite de l'instruction INLINE. Après lancement du programme, le débogueur annonce une *"Illegal Instruction"*. Vous pouvez alors faire désassembler ou éditer la zone désignée par d0. Vous pouvez ensuite faire reprendre l'exécution du programme BASIC avec l'instruction GO du débogueur après avoir éventuellement appuyé sur *Shift-Alternate-Control*.

CALL adr([x,y,...])

```
adr : avar (au moins 32 bits, de préférence adr%)
x,y : iexp
```

L'instruction CALL permet d'appeler des sous-programmes en assembleur ou en C. *'adr'* représente ici l'adresse à partir de laquelle le programme assembleur figure dans la mémoire. Il est possible de transmettre une liste de paramètres.

Après appel de CALL, l'adresse de retour figure sur la pile (la routine assembleur doit se terminer par RTS). Vient ensuite le nombre de paramètres transmis sous forme d'une valeur 16 bits et enfin l'adresse, sous forme d'une valeur 32 bits, à partir de laquelle les paramètres figurent dans la mémoire. Tous les paramètres sont interprétés comme des longs mots (32 bits).

Il est également possible ici de transmettre des chaînes de caractères comme paramètres. Dans ce cas, la valeur transmise est l'adresse de départ de la chaîne de caractères.

Structure de la pile

```
 (sp) -->  Adresse de retour
4(sp) -->  Nombre de paramètres (16 bits)
6(sp) -->  Adresse du tableau de paramètres (32 bits)
```

RCALL adr,reg%()

```
reg%() : nom d'un tableau entier de 4 octets
adr : iexp
```

L'instruction RCALL permet de prédéfinir les registres avec certaines valeurs avant de lancer la routine assembleur puis de tester le contenu des registres après exécution de la routine. On utilise à cet effet le tableau *reg%()*, dont les éléments doivent être du type entier de 4 octets et qui doit compter au moins 16 éléments. Avant le lancement de la routine assembleur, les entrées de ce tableau sont copiées dans les registres et après exécution de la routine le contenu des registres est écrit dans les éléments correspondants du tableau, d'après les conventions suivantes (pour OPTION BASE 0) :

```
Registres de données      d0 à d7      dans reg%(0)      à   reg%(7)
Registres d'adresse       a0 à a6      dans reg%(8)      à   reg%(14)
User Stack Pointer        (a7)         dans reg%(15)     (retour uniquement)
```

Exemple :

Le listing assembleur attend dans a0 l'adresse de la mémoire écran (logique ou physique). Il inverse ensuite l'écran entre les coordonnées y transmises dans d0 et d1. Les coordonnées sont multipliées par deux sur l'écran couleur.

```
sub    d0,d1      ;Nombre de lignes
mulu   #20,d1     ;Critère d'arrêt de la boucle
subq   #1,d1      ; "
mulu   #80,d0     ;Nombre d'octets à sauter
add.l  d0,a0      ;Adresse à partir de laquelle inverser
```

```
boucle              ;Début de la boucle
 not.l (a0)+        ;Inverser 4 octets
 dbra d1,boucle     ;Décrémenter et sauter
 rts                ;Retour à GFA-BASIC
```

Le programme GFA-BASIC d'appel se présente ainsi :

```
DO
  READ a%
  EXIT IF a%=-1
  A$=A$+MKI$(a%)
LOOP
DATA 37440,49916,20,21313
DATA 49404,80,53696
DATA 18072,20937,65532,20085,-1
'
DIM r%(16)
xb2%=XBIOS(2)
HIDEM
FOR j%=1 TO 50
  FOR i%=0 TO 190 STEP 10
    r%(0)=i%
    r%(1)=399-i%
    r%(8)=xb2%
    RCALL V:a$,r%()
  NEXT i%
NEXT j%
```

--> Ce programme produit un jeu graphique. Il est naturellement également possible ici, comme pour C:, d'utiliser INLINE.

EXEC mod,nom,cmdl,envs
EXEC(mod,nom,cmdl,envs)

```
mod : iexp
nom,cmdl,envs : sexp
```

L'instruction EXEC peut être employée comme instruction mais aussi comme fonction. Elle sert à charger et lancer des programmes à partir de la disquette qui rendront le contrôle au programme d'appel après avoir été exécutés. Avant d'appeler EXEC, il faut mettre la place mémoire nécessaire à la disposition du programme d'appel (voyez l'exemple). Le paramètre *mod* spécifie le mode de l'appel, d'après les conventions suivantes :

```
0 --> Charger et lancer le programme
```

```
3 --> Seulement charger le programme
```

L'expression *nom* contient le nom du programme à charger (et à lancer). Le format de la spécification du nom obéit aux règles du système de fichiers hiérarchisé telles qu'elles ont été décrites dans le chapitre sur les entrées et sorties générales.

L'expression *cmdl* contient la ligne d'instruction qui est inscrite dans la Base Page du programme appelé. Le premier caractère de la ligne d'instruction contient sa longueur (127 au maximum). Cet octet est en général ignoré sauf s'il vaut zéro. C'est pourquoi on pourra normalement se permettre d'y placer un caractère dummy (*).

envs contient l'environnement. Il s'agit d'une chaîne subdivisée à l'aide de CHR$(0). Pour un programme en C, il y aura donc une série de chaînes dont la dernière sera marquée par un double octet nul (GFA-BASIC s'en charge automatiquement). Cet environnement est utilisé par de nombreux programmes Shell pour sauvegarder des variables et leurs noms (généralement l'instruction SET). De nombreux compilateurs l'utilisent également pour déterminer les chemins d'accès etc...

Lorsque EXEC est appelée comme fonction, on obtient en retour la valeur renvoyée par le programme ou, si mod=3, l'adresse de la Base Page du programme appelé. EXEC 3 ne peut être utilement employée qu'avec des programmes écrits spécialement à cet effet (overlays) ou avec le débogueur.

Exemple :

```
FILESELECT "\*.PRG","",f$
RESERVE 100
SHOWM
a%=EXEC 0,f$,"",""
RESERVE
PRINT "Retour à GFA-BASIC", valeur de réponse",a%
```

--> Permet d'appeler un programme (s'il ne nécessite pas trop de place mémoire) et retourne à l'interpréteur une fois le programme appelé achevé. La valeur en réponse fournie par la fonction EXEC peut également être donnée à l'aide des instructions QUIT n et SYSTEM n de GFA-BASIC.

11. BIBLIOTHEQUES AES

Le présent chapitre a pour objet de vous fournir un aperçu des bibliothèques AES (Application Environment Services). Il ne nous serait toutefois pas possible de décrire de façon exhaustive ces routines dans le cadre de ce manuel. Nous ne pouvons donc que vous renvoyer à la littérature très complète consacrée à GEM. Nous ne vous fournirons donc dans ce chapitre que des informations sous une forme condensée. Les différentes routines sont présentées suivant le schéma suivant :

Sur la première ligne figure le nom des routines de bibliothèques, suivi d'une brève description de la fonction. Vient ensuite l'appel de fonction complet en notation GFA-BASIC 3.0 et une description du rôle de toutes les variables utilisées.

Le chapitre se terminera par quelques programmes d'exemple très complets.

Avant de commencer la description des instructions des 11 bibliothèques AES, nous devons présenter les principales structures de données utilisées par l'AES. Il s'agit des structures OBJECT, TEDINFO, ICOBLK, BITBLK, USERBLK et PARMBLK.

Nous vous vous proposons tout d'abord une présentation des instructions AES qui se présentent sous une forme proche de celles dont disposait déjà GFA-BASIC 2.0.

GCONTRL
ADDRIN
ADDROUT
GINTIN
GINTOUT
GB

Ces fonctions déterminent les adresses des blocs de paramètres AES :

```
GCONTRL --> Adresse du tableau de contrôle AES
ADDRIN  --> Adresse du tableau d'entrée d'adresse AES
ADDROUT --> Adresse du tableau de sortie d'adresse AES
GINTIN  --> Adresse du tableau d'entrée entière AES
GINTOUT --> Adresse du tableau de sortie entière AES
GB      --> Adresse du tableau de paramètres AES
```

Avec l'index entre parenthèses, ces fonctions accèdent directement aux champs appropriés. GCONTRL(3) équivaut à DPEEK(GCONTRL+6) ou au POKE correspondant.

GCONTRL, GINTIN et GINTOUT sont organisés par mots, ADDRIN et ADDROUT par longs mots.

ADDRIN(2)=x correspond donc à LPOKE ADDRIN+2*4,x.

Le tableau GB ne peut être utilisé avec des indices car il n'a été prévu que par souci de compatibilité avec le ST-BASIC ; c'est pourquoi vous disposez de GINTIN etc... Le second long mot ({GB+4}) est l'adresse du tableau global interne à GEM.

Exemple :

voyez GEMSYS.

GEMSYS no

```
no : iexp
```

L'instruction GEMSYS permet d'appeler une routine AES à l'aide de son numéro de fonction. Les paramètres nécessaires à cette routine doivent être placés dans les blocs de paramètres AES.

Exemple :

```
REPEAT
  GINTIN(0)=60
  GINTIN(1)=30
  GINTIN(2)=10
  GINTIN(3)=10
  GINTIN(4)=200
  GINTIN(5)=200
  GEMSYS 72
UNTIL MOUSEK
```

--> Dessine un rectangle se déplaçant sur l'écran (72 est le numéro de la fonction GRAF_MOVEBOX).

Structure d'objet

Offset	Contenu	Type	Signification
00	ob_next	word	Pointeur sur l'objet suivant
02	ob_head	word	Pointeur sur le premier enfant
04	ob_tail	word	Pointeur sur le dernier enfant
06	ob_type	word	Type de l'objet
08	ob_flags	word	Informations d'objet (voir plus bas)
10	ob_state	word	Etat de l'objet (voir plus bas)
12	ob_spec	long	Pointeur sur des informations supplémentaires (voir plus bas)
16	ob_x	word	Position X de l'objet
18	ob_y	word	Position Y de l'objet
20	ob_w	word	Largeur de l'objet
22	ob_h	word	Hauteur de l'objet

Un objet occupe donc en mémoire ces 24 octets plus les structures de description supplémentaires éventuelles, telles que des structures TEDINFO ou BITBLK par exemple.

OB_NEXT Pointeur sur l'objet suivant de même niveau ou sur l'objet-père s'il s'agit du dernier élément pour ce niveau.

OB_HEAD Pointeur sur le premier enfant de l'objet correspondant.

OB_TAIL Pointeur sur le dernier enfant de l'objet correspondant.

S'il n'y a pas de niveaux plus bas, les pointeurs présentent la valeur -1, également dite NIL (Not In List).

Les valeurs OB_TYPE suivantes sont possibles. Suivant la valeur de OB_TYPE, OB_SPEC représentera l'adresse de différentes structures d'informations.

OB_TYPE		OB_SPEC
G_TEXT	21	Adresse d'une structure TEDINFO
G_BOXTEXT	22	Adresse d'une structure TEDINFO
G_IMAGE	23	Adresse d'une structure BITBLK
G_USERDEF	24	Adresse d'une structure USERBLK
G_IBOX	25	BOXINFO, voir plus bas
G_BUTTON	26	Adresse de la chaîne contenant le texte
G_BOXCHAR	27	BOXINFO, voir plus bas
G_STRING	28	Adresse de la chaîne correspondante
G_FTEXT	29	Adresse d'une structure TEDINFO
G_FBOXTEXT	30	Adresse d'une structure TEDINFO
G_ICON	31	Adresse d'une structure ICONBLK
G_TITLE	32	Adresse de la chaîne correspondante

Pour G_BOX, G_IBOX et G_BOXCHAR, OB_SPEC contient des informations concernant les caractères, le cadre et la couleur de l'objet correspondant. Les 8 bits supérieurs ne sont définis que pour G_BOXCHAR où ils contiennent le code ASCII du caractère apparaissant dans le sélecteur.

Ils contiennent les valeurs suivantes pour le cadre :

```
        0 =     pas de cadre
  1 à 128 =     le bord se situe 1 à 128 points écran à l'intérieur de l'objet
                correspondant.
-1 à -128 =     le bord se situe 1 à 128 points écran en dehors de l'objet
                correspondant.
```

Définition des bits pour la couleur de l'objet : 1111 2222 3444 5555

```
1   Cadre                       (0 à 15)

2   Texte                       (0 à 15)

3   Texte                       (0 = transparent, 1 = couvrant)

4   Motif de remplissage        (0 à 7)

5   Couleur de l'intérieur de l'objet
```

OB_FLAGS	Hexa	Numéro de bit
SELECTABLE	&H0001	0
DEFAULT	&H0002	1
EXIT	&H0004	2
EDITABLE	&H0008	3
RBUTTON	&H0010	4
LASTOB	&H0020	5
TOUCHEXIT	&H0040	6
HIDETREE	&H0080	7
INDIRECT	&H0100	8

OB_STATE	Hexa	Numéro de bit
SELECTED	&H0001	0
CROSSED	&H0002	1
CHECKED	&H0004	2
DISABLED	&H0008	3
OUTLINED	&H0010	4
SHADOWED	&H0020	5

Les structures décrites dans la section précédente sont appelées sous GFA-BASIC 3.0 avec la syntaxe suivante (lecture et écriture) :

OB_NEXT(tree%,obj&)
OB_HEAD(tree%,obj&)
OB_TAIL(tree%,obj&)

OB_TYPE(tree%,obj&)
OB_FLAGS(tree%,obj&)
OB_STATE(tree%,obj&)
OB_SPEC(tree%,obj&)

OB_X(tree%,obj&)
OB_Y(tree%,obj&)
OB_W(tree%,obj&)
OB_H(tree%,obj&)

tree% représente ici l'adresse de l'arbre d'objets et *obj&* le numéro de l'objet pour lequel des informations sont attendues.

L'adresse de l'objet peut également être obtenue avec
OB_ADR(tree%,obj&)

Structure d'informations de texte (TEDINFO)

Offset	Contenu	Type	Signification
00	te_ptext	long	Pointeur sur le texte
04	te_ptmplt	long	Pointeur sur le masque de texte
08	te_pvalid	long	Pointeur sur la chaîne définissant les caractères admis en entrée
12	te_font	word	Jeu de caractères
14	te_resvd	word	Réservé
16	te_just	word	Orientation du texte
18	te_color	word	Couleur du sélecteur d'encadrement
20	te_resvd2	word	Réservé
22	te_thickness	word	Epaisseur du cadre
24	te_txtlen	word	Longueur du texte
26	te_tmplen	word	Longueur du masque de texte

Structure d'icône (ICONBLK)

Offset	Contenu	Type	Signification
00	ib_pmask	long	Pointeur sur le masque d'icône
04	ib_pdata	long	Pointeur sur les données d'icône
08	ib_ptext	long	Pointeur sur le texte d'icône
12	ib_char	word	Caractère à l'intérieur de l'icône
14	ib_xchar	word	Position x du caractère
16	ib_ychar	word	Position y du caractère
18	ib_xicon	word	Position x de l'icône
20	ib_yicon	word	Position y de l'icône
22	ib_wicon	word	Largeur de l'icône
24	ib_hicon	word	Hauteur de l'icône
26	ib_xtext	word	Position x du texte
28	ib_ytext	word	Position y du texte
30	ib_wtext	word	Largeur du texte en points écran
32	ib_htext	word	Hauteur du texte en points écran
34	ib_resvd	word	Réservé

Structure d'image de bits (BITBLK)

Offset	Contenu	Type	Signification
00	bi_pdata	long	Pointeur sur les données d'image
04	bi_wb	word	Largeur de l'image en octets
06	bi_hl	word	Hauteur de l'image en points écran
08	bi_x	word	Position x de l'image
10	bi_y	word	Position y de l'image
12	bi_color	word	Couleur de l'image

Structure de bloc d'application (USERBLK)

Offset	Contenu	Type	Signification
00	ub_code	long	Pointeur sur une routine d'affichage de l'objet définie par l'utilisateur
04	up_parm	long	Pointeur sur une structure PARMBLK

Il doit s'agir ici de routines assembleur.

Structure de bloc de paramètres (PARMBLK)

Offset	Contenu	Type	Signification
00	pb_tree	long	Pointeur sur l'arbre d'objets
04	pb_obj	word	Numéro d'objet
06	pr_prevstate	word	Etat précédent
08	pr_currstate	word	Etat actuel
10	pb_x	word	Position x de l'objet
12	pb_y	word	Position y de l'objet
14	pb_w	word	Largeur
16	pb_h	word	Hauteur
18	pb_xc	word	Position x du rectangle de Clipping
20	pb_yc	word	Position y du rectangle de Clipping

22	pb_wc	word	Largeur du rectangle de Clipping
24	pb_hc	word	Hauteur du rectangle de Clipping
28	pb_parm	long	Paramètres de la structure USERBLK

Dans le domaine des fonctions AES, il convient de noter que certaines fonctions ne renvoient pas seulement une valeur de fonction mais renvoient aussi des valeurs de réponse dans des variables préparées à cet effet. D'autres fonctions renvoient des valeurs réservées et peuvent être appelées à l'aide de VOID ou de son abréviation, le tilde "~". Lorsque des adresses sont spécifiées, il faut employer des types de variables de 4 octets de long au moins et les indications de coordonnées doivent être effectuées à l'aide de types de variables de 2 octets de long au moins.

Bibliothèque d'application (application library)

Gère les accès à d'autres bibliothèques AES. Les fonctions APPL_INIT et APPL_EXIT sont automatiquement appelées par GFA-BASIC 3.0 lors du lancement ou de l'abandon du programme.

APPL_INIT()

Le programme actuel est déclaré comme application GEM. La fonction APPL_INIT fournit le handle du programme GFA-BASIC actuel. Ce numéro d'identification est très important par exemple pour l'installation de fontes supplémentaires.

reponse contient le numéro d'identification (*ap_id*) de l'application (du programme) pour GEM. APPL_INIT est utilisée pour APPL_READ, APPL_WRITE et MENU_REGISTER. Sous GFA-BASIC, cette fonction est une fonction fictive qui n'effectue aucun appel du système d'exploitation car APPL_INIT est exécutée dès le lancement de l'interpréteur GFA-BASIC. C'est donc la *ap_id* obtenue à cette occasion qui sera renvoyée.

APPL_READ (id,len,adr_buffer)

```
id,len,adr_buffer : iexp
```

Cette instruction permet de lire des octets dans un buffer d'événement (message buffer).

reponse 0 = Erreur apparue
id Numéro de l'application dont le buffer est lu
len Nombre d'octets à lire
adr_buffer Adresse à partir de laquelle les octets lus sont stockés

APPL_WRITE(id,len,adr_buffer)

```
id,len,adr_buffer : iexp
```

Ecrit des octets dans un buffer d'événement.

reponse 0 = Erreur apparue
id Numéro de l'application dans le buffer de laquelle il s'agit d'écrire
len Nombre d'octets à écrire
adr_buffer Adresse à partir de laquelle les octets sont écrits

APPL_FIND(fname$)

```
fname$ : sexp
```

Détermine le numéro d'identification d'une application dont le nom de fichier est connu, par exemple afin de permettre l'échange d'informations avec d'autres programmes en cours d'exécution.

reponse Identification du programme recherché
-1 = Erreur apparue, programme non trouvé
fname contient le nom de fichier, long de 8 caractères (sans extension) de l'application à rechercher. Le nom de fichier doit absolument avoir une longueur de 8 caractères, être écrit en majuscules, et doit donc éventuellement être comblé avec des espaces.

Exemple :

```
PRINT APPL_FIND("CONTROL")
```

--> Renvoie 2 (à peu près) si l'accessoire de contrôle est chargé ou -1 s'il ne l'est pas. Outre les accessoires, on peut faire rechercher le programme utilisateur "GFABASIC" (ap_id=0) et le gestionnaire d'écran "SCREENMGR" (ap_id=1) qui se charge par exemple de la gestion des menus.

APPL_TPLAY(mem,num,scale)
APPL_TRECORD(mem,num)

```
mem,num,scale : iexp
```

Ces instructions sont censées permettre d'enregistrer et de reproduire une trace des activités de l'utilisateur (souris, touches actionnées et intervalles entre les diverses actions) un peu comme un magnétophone avec ses fonctions *Play* et *Record*. Ces fonctions ne fonctionnent cependant pas conformément à la description fournie dans la documentation de GEM.

Avec les nouvelles versions de la ROM, ces fonctions fonctionnent presque comme indiqué (8 octets sont employés au lieu de 6 pour enregistrer un événement et le facteur d'échelle scale fonctionne également différemment) mais comme leur comportement apparaît très peu fiable, il est tout à fait déconseillé d'y avoir recours. C'est pourquoi nous ne les décrirons pas plus avant.

APPL_EXIT()

APPL_EXIT annule la déclaration d'un programme auprès de l'AES. Le numéro d'identification du programme voulu est libéré et redevient ainsi disponible pour d'autres programmes.

APPL_EXIT n'existe sous GFA-BASIC qu'en tant que fonction fictive car une instruction QUIT/SYSTEM effectue automatiquement cette opération.

reponse 0 = Erreur apparue

Bibliothèque d'événements (event library)

Réagit aux entrées à l'aide de la souris ou du clavier ainsi qu'aux événements liés au temps.

EVNT_KEYBD()

Attend un événement du clavier et renvoie le code clavier correspondant.

reponse Code touche (low byte) et code clavier (high byte) de la touche enfoncée

Exemple :

```
DO
  PRINT HEX$(EVNT_KEYBD(),4)
LOOP
```

--> Sort le code des touches pressées en hexadécimal.

EVNT_BUTTON(clicks,mask,state [,mx,my,button,k_state])

```
clicks,mask,state : iexp
mx,my,button,k_state : ivar
```

Attend et détermine un événement des boutons de la souris.

reponse	Nombre de fois que les boutons de la souris ont été actionnés
clicks	Nombre de clics nécessaires sur les boutons
mask	Masque du bouton de la souris voulu Bit 0 = gauche, bit 1 = droite
state	Etat voulu pour déclencher l'événement Même définition des bits que pour mask
mx	Coordonnée x lors du déclenchement de l'événement par le curseur de la souris
my	Coordonnée y lors du déclenchement de l'événement par le curseur de la souris
button	Etat des boutons de la souris, comme pour state
k_state	Etat des touches de commutation du clavier lors du déclenchement de l'événement : Bit 0 = touche Shift de droite Bit 1 = touche Shift de gauche Bit 2 = touche Control Bit 3 = touche Alternate

Les paramètres *mx,my,button* et *k_state* sont optionnels. Ces valeurs peuvent aussi être testées à l'aide de GINTOUT(1) à GINTOUT(4).

Exemple :

```
DO
  SELECT EVNT_BUTTON(2,1,1,mx%,my%,bu%,kb%)
  CASE1
    PLOT mx%,my%
  CASE2
    PBOX mx%-10,my%-10,mx%+10,my%+10
```

```
    ENDSELECT
  LOOP UNTIL BTST(kb%,2)
```

--> Attend des clics sur le bouton gauche de la souris. Un point est fixé pour un clic simple et un carré pour un double clic. Cet exemple se termine lorsque la touche *Control* est appuyée en plus du clic.

EVNT_MOUSE(flags,mx,my,mw,mh,mcur_x,mcur_y,button,k_state)

```
flags,mx,my,mw,mh : iexp
mcur_x,mcur_y,button,k_state : ivar
```

Attend que le pointeur de la souris pénètre ou sorte d'une zone rectangulaire de l'écran.

reponse	Réservé, toujours 1
flags	Annonce que la zone rectangulaire de l'écran a été pénétrée (0) ou abandonnée (1)
mx	Coordonnée gauche x
my	Coordonnée supérieure y
mw	Largeur
mh	Hauteur de la zone rectangulaire de l'écran
mcur_x	Coordonnée x
mcur_y	Coordonnée y du curseur de la souris
button	Etat des boutons de la souris lors de l'intervention de l'événement Bit 0 = bouton droit de la souris Bit 1 = bouton gauche de la souris
k_state	Etat du clavier lors de l'intervention de l'événement Bit 0 = touche Shift de droite Bit 1 = touche Shift de gauche Bit 2 = touche Control Bit 3 = touche Alternate

Exemple :

```
DO
  EVNT_MOUSE(0,100,100,200,90,mx%,my%,bu%,kb%)
  IF bu%
    PBOX mx%-10,my%-10,mx%+10,my%+10
  ELSE
    PLOT mx%,my%
  ENDIF
LOOP UNTIL BTST(kb%,2)
```

--> On attend que le curseur de la souris se trouve dans un rectangle. Dès que c'est le cas, un rectangle est dessiné si un bouton de la souris est actionné, sinon un point. Pour terminer, appuyez sur *Control* lorsque le pointeur de la souris figure à l'intérieur d'un rectangle.

Attention : il est possible d'appuyer sur *Shift-Control-Alternate* pendant l'exécution de routines GEM mais cela ne produira d'effet qu'après le retour en BASIC.

-EVNT_TIMER(3600000) ou DELAY 3600 produiraient par exemple une pause d'une heure avant que l'ordinateur revienne de GEM.

EVNT_MESAG(adr_buffer)

```
adr_buffer : iexp
```

Attend l'intervention d'un message dans le buffer d'événement.

reponse Réservé, toujours 1
adr_buffer Adresse du buffer de 16 octets de long (message buffer) servant à recevoir le message (voyez MENU(1) à MENU(8))

Si 0 est spécifié pour adr_buffer, c'est automatiquement le buffer de message propre à GFA-BASIC qui sera utilisé, c'est-à-dire MENU(1) à MENU(8).

EVNT_TIMER(count)

```
count : iexp
```

Attend l'écoulement d'un délai à spécifier en millisecondes (voyez également DELAY).

reponse Réservé, toujours 1
count Nombre de millisecondes

EVNT_MULTI(flags,clicks,mask,state,m1_flags,m1_x,m1_y,m1_w, m1_h,m2_flags,m2_x,m2_y,m2_w,m2_h,adr_buffer, count [,mcur_x,mcur_y,button,k_state,key,nmb_clicks])

```
flags,clicks,mask,state,m1_flags,m1_x,m1_y,m1_w,m1_h : iexp
m2_x,m2_y,m2_w,m2_h,adr_buffer, count : iexp
mcur_x,mcur_y,button,k_state,key,nmb_clicks : ivar
```

Attend l'intervention de différents événements.

reponse Contient l'événement s'étant effectivement produit, définition des bits comme pour *ev_mflags*.

flags Fixe l'événement que l'application devra attendre, avec les conventions suivantes :

Bit 0	Clavier	MU_KEYBD
Bit 1	Bouton de la souris	MU_BUTTON
Bit 2	Premier évènement souris	MU_M1
Bit 3	Second évènement souris	MU_M2
Bit 4	Evènement de communication	MU_MESAG
Bit 5	Timer	MU_TIMER

nmb_clicks Nombre de fois que les boutons de la souris ont été actionnés.

Les paramètres suivants ont déjà été expliqués pour EVNT_MOUSE, EVNT_KEYBD, EVNT_BUTTON et EVNT_MESAG. Il convient cependant de noter qu'il est possible d'attendre deux événements souris différents (m1 et m2). Avec ON MENU, qui utilise cette routine en interne, les paramètres sont fixés avec ON MENU xxx GOSUB ou bien *count* est spécifié directement dans l'instruction.

```
MENU(1) à MENU(8)       Buffer de message
MENU(9)                 Valeur de réponse.
MENU(10) = mcur_x       Position du curseur de la souris
MENU(11) = mcur_y       Position du curseur de la souris
MENU(12) = button       Bouton de la souris
MENU(13) = k_state      BIOS(11,-1)
MENU(14) = key          Codes clavier et ASCII
MENU(15) = nmb_clics    Nombre de clics de la souris
```

EVNT_DCLICK(nouv,get_set)

```
nouv,get_set : iexp
```

Fixe la vitesse des doubles clics de la souris.

reponse Ancienne vitesse
nouv Nouvelle vitesse (0 à 4)
0 = déterminer valeur actuelle sans tenir compte de nouv
1 = fixe la nouvelle valeur pour nouv

Bibliothèques des menus (menu library)

Se charge du dessin et de la gestion d'une barre de menus.

MENU_BAR(tree,flag)

```
tree,flag : iexp
```

Active ou désactive une barre de menus (tirée d'un fichier Resource).

Cf. MENU x$() et MENU KILL.

reponse 0 = Erreur apparue
tree Adresse de l'arbre d'objets du menu
flag 1 = activer barre de menus
0 = désactiver barre de menus (toujours en fin de programme)

MENU_ICHECK(tree,item,flag)

```
tree,item,flag : iexp
```

Fixe ou efface un crochet devant une entrée du menu (c'est pourquoi il est conseillé de réserver au moins deux espaces à cet effet devant chaque entrée de menu).

Cf. MENU x,0 et MENU x,1.

reponse 0 = Erreur apparue
tree Adresse de l'arbre d'objets du menu

item Numéro d'objet de l'entrée du menu correspondante
flag 0 = fixer crochet
1 = effacer crochet

MENU_IENABLE(tree,item,flag)

```
tree,item,flag : iexp
```

Active ou désactive des entrées du menu. L'entrée de menu voulue est alors affichée en caractères gris clairs.

Cf. MENU x,2 et MENU x,3.

reponse 0 = Erreur apparue
tree Adresse de l'arbre d'objets du menu
item Numéro d'objet de l'entrée correspondante du menu
flag 0 = désactiver
1 = activer

MENU_TNORMAL(tree,title,flag)

```
tree,title,flag : iexp
```

Commute les titres de menus sur l'affichage inversé ou normal.

Cf. MENU OFF.

reponse 0 = Erreur apparue
tree Adresse de l'arbre d'objets du menu
title Numéro d'objet du titre de menu correspondant
flag 0 = Affichage inversé
1 = Affichage normal

MENU_TEXT(tree,item,nouv_texte$)

```
tree,item : iexp
nouv_texte$ : sexp
```

Modifie le texte d'une entrée du menu. Cette fonction permet d'adapter des entrées du menu à la situation actuelle du programme.

reponse 0 si une erreur est apparue
tree Adresse de l'arbre d'objets du menu
item Numéro de l'entrée d'objet à modifier
nouv_texte$ Chaîne contenant la nouvelle entrée du menu
(ne doit pas dépasser la longueur de l'ancienne entrée)

MENU_REGISTER(ap_id,m_text$)

```
ap_id : iexp
m_text$ : sexp
```

Contient les noms d'un accessoire sous le premier titre de menu. 6 entrées de menu-accessoires peuvent êtres mises en place au maximum.

Cette fonction ne peut toutefois être employée utilement que par un accessoire (seulement donc dans une version compilée du programme).

reponse Code de l'entrée du menu de l'accessoire correspondant
0 à 5 pour les première à sixième entrées d'accessoire,
-1 lorsqu'aucune autre entrée n'est plus possible
ap_id Numéro d'identification de l'accessoire correspondant
m_text$ Adresse de l'entrée correspondante du menu

Bibliothèque d'objets (object library)

Contient des routines servant à la définition, à l'affichage et à la modification d'objets.

OBJC_ADD(tree,parent,child)

```
tree,parent,child : iexp
```

Insère un objet à l'intérieur d'un arbre d'objets déterminé et établit la liaison entre les objets préexistants et le nouvel objet.

reponse 0 = Erreur apparue
tree_adr Adresse de l'arbre d'objets correspondant
parent Numéro d'objet de l'objet-"père" auquel vient s'adjoindre le nouvel objet
child Numéro d'objet de l'objet-"fils" adjoint à l'objet-"père"

OBJC_DELETE(tree,del_obj)

```
tree,del_obj : iexp
```

Modifie les pointeurs *Next, Head* et *Tail* d'un objet à l'intérieur de l'arbre d'objets correspondant, de sorte qu'il ne puisse plus être sélectionné jusqu'à ce que ces pointeurs soient à nouveau restaurés.

reponse	0 = Erreur apparue
tree	Adresse de l'arbre d'objets correspondant
del_obj	Numéro d'objet de l'objet à supprimer

OBJC_DRAW(tree,start_obj,depth,cx,cy,cw,ch)

```
tree,start_obj,depth,cx,cy,cw,ch : iexp
```

Cette fonction affiche sur l'écran des objets entiers ou des parties d'objets. En option peuvent être spécifiées les coordonnées d'une zone rectangulaire de l'écran à laquelle l'affichage sera limité.

reponse	0 = Erreur apparue
tree	Adresse de l'arbre d'objets correspondant
start_obj	Numéro du premier objet à afficher 0 = ne dessiner que le premier objet
depth	Nombre de niveaux d'objets à afficher
cx	Coordonnée gauche x
cy	Coordonnée supérieure y
cw	Largeur et
ch	Hauteur du rectangle de délimitation

OBJC_FIND(tree,start_obj,depth,fx,fy)

```
tree,start_obj,depth : iexp
fx,fy : ivar
```

Cette fonction permet d'indiquer des coordonnées du curseur de la souris pour obtenir le numéro de l'objet figurant dans l'emplacement de l'écran spécifié.

reponse	Numéro de l'objet trouvé -1 = aucun numéro d'objet identifié
tree	Adresse de l'arbre d'objets à examiner

start_obj	Numéro du premier objet à examiner
depth	Nombre de niveaux d'objets à examiner 0 = uniquement l'objet-"père"
fx	Coordonnée x et
fy	Coordonnée y de l'emplacement de l'écran à spécifier

OBJC_OFFSET(tree,obj,x_abs,y_abs)

```
tree,obj : iexp
x_abs,y_abs : ivar
```

Calcule les coordonnées absolues d'un objet spécifié sur l'écran.

reponse	0 = Erreur apparue
tree	Adresse de l'arbre d'objets à spécifier
obj	Numéro de l'objet à spécifier
x_abs	Les coordonnées absolues x
y_abs	et y identifiées

OBJC_ORDER(tree,obj,nouv_pos)

```
tree,obj,nouv_pos : iexp
```

Modifie l'ordre d'un objet à spécifier à l'intérieur des niveaux d'objets.

reponse	0 = Erreur
tree	Adresse de l'arbre d'objets voulu
obj	Numéro de l'objet à modifier
nouv_pos	Numéro du niveau d'objet -1 = niveau supérieur 0 = niveau le plus bas 1 = niveau le plus bas + 1 2 = niveau le plus bas + 2 etc...

OBJC_EDIT(tree,obj,char,anc_pos,flag,nouv_pos)

```
tree,obj,char,anc_pos,flag : iexp
nouv_pos : ivar
```

Permet d'entrer et d'éditer des textes dans les types d'objets G_TEXT et G_BOXTEXT.

reponse	0 = Erreur apparue
tree	Adresse de l'arbre d'objets voulu
obj	Numéro de l'objet voulu
char	Caractère entré (y compris code clavier)
old_pos	Position du caractère dans la chaîne en entrée
flag	Fonction sélectionnée

```
0  ED_START =    Réservé - appel fictif
1  ED_INIT  =    Calculer chaine formatée et activer curseur
2  ED_CHAR  =    Traiter caractère voulu et afficher la chaîne
                 à nouveau
3  ED_END   =    Désactiver curseur
```

nouv_pos	Nouvelle position du caractère dans la chaîne d'entrée

OBJC_CHANGE(tree,obj,res,cx,cy,cw,ch,nouv_etat,re_draw)

```
tree,obj,res,cx,cy,cw,ch,nouv_etat,re_draw : iexp
```

Cette fonction modifie l'état d'un objet (OB_STATE) et inverse le cas échéant la partie de l'objet située à l'intérieur de la zone de délimitation Clip. Normalement, OB_STATE est modifié directement, puis OBJC_DRAW est appelé.

reponse	0 = Erreur
tree	Adresse de l'arbre d'objets correspondant
obj	Numéro de l'objet à modifier
res	Réservé, toujours 0
cx	Coordonnée gauche x
cy	Coordonnée supérieure y
cw	Largeur et
ch	Hauteur du rectangle de délimitation
nouv_etat	Nouvel état de l'objet (voyez OB_STATE)
re_draw	0 = ne pas redessiner l'objet
	1 = redessiner l'objet

Bibliothèque de formulaires (form library)

Contient des routines de gestion de formulaires.

FORM_DO(tree,start_obj)

```
tree,start_obj : iexp
```

Cette fonction se charge de la gestion complète d'un objet de formulaire jusqu'à ce qu'un objet ait été cliqué avec l'état EXIT ou TOUCH_EXIT Etat.

reponse	Numéro de l'objet cliqué pour terminer (en cas de double clic, le 15ème bit est mis).
tree	Adresse de l'arbre d'objets correspondant
start_obj	Numéro du premier objet dans l'arbre d'objets

FORM_DIAL(flag,mi_x,mi_y,mi_w,mi_h,ma_x,ma_y,ma_w,ma_h)

```
flag,mi_x,mi_y,mi_w,mi_h,ma_x,ma_y,ma_w,ma_h : iexp
```

Cette fonction sert à réserver ou à libérer des zones rectangulaires de l'écran et à dessiner des rectangles en contraction ou en expansion.

reponse 0 si erreur apparue

flag Type de fonction :

```
0 = FMD_START     réserve une zone de l'écran
1 = FMD_GROW      dessine un rectangle en expansion
2 = FMD_SHRINK    dessine un rectangle en contraction
3 = FMD_FINISH    libère à nouveau zone d'écran réservée
```

mi_x, *mi_y*, *mi_w*, *mi_h* Coordonnées du rectangle en expansion minimale

ma_x, *ma_y*, *ma_w*, *ma_h* Coordonnées du rectangle en expansion maximale

Explication : FORM_DIAL(1,0,0,0,0,100,100,300,100) dessine un rectangle en expansion. 0,0,0,0 signifie ici que le rectangle apparaîtra au centre du rectangle de destination.

FORM_ALERT(button,string$)

```
button : iexp
string$ : sexp
```

Affiche un message d'avertissement d'ordre général (ALERT_BOX).

reponse Numéro du bouton avec lequel le sélecteur d'alert a été abandonné
button Numéro du bouton par défaut
0 = aucun
1 = le premier
2 = le second
3 = le troisième
string$ Adresse de la chaîne contenant le message.
La chaîne présente le format suivant :

```
[i][message][button]
```

avec les définitions suivantes :

```
i          un chiffre exactement     0 = aucun icône
                                     1 = point d'exclamation
                                     2 =point d'interrogation
                                     3 = panneau Stop
message    5 lignes de texte de 30 caractères chacune au
           maximum, les lignes étant séparées entre elles par |
button     3 noms de boutons au maximum, séparés entre eux par |
```

Exemple :

```
-FORM_ALERT(1,ERR$(100))
```

FORM_ERROR(err)

```
err : iexp
```

Affiche un message d'avertissement en cas d'erreurs MS-DOS.

reponse Numéro du bouton avec lequel le message d'avertissement a été validé.
err Numéro d'erreur MS-DOS.

FORM_CENTER(tree,fx,fy,fw,fh)

```
tree : iexp
fx,fy,fw,fh : ivar
```

Cette fonction centre les coordonnées d'objet spécifiées.

onse Réservé, toujours 1
Adresse de l'arbre d'objets voulu
Coordonnée gauche x
Coordonnée supérieure y
Largeur et
h hauteur du rectangle centré

FORM_KEYBD(tree,obj,next_obj,char,nouv_obj,next_char)

```
tree,obj,next_obj,char,nouv_obj : iexp
next_char : ivar
```

Permet les entrées au clavier dans un formulaire (voyez OBJC_EDIT).

reponse 0 = objet avec état EXIT actionné
supérieur à 0 = dialogue pas encore terminé
tree Adresse de l'arbre d'objets correspondant
obj Numéro de l'objet EDIT actuel
next_obj Numéro du prochain objet EDIT
char Caractère entré
nouv_obj Nouvel objet EDIT pour le prochain appel
next_char Prochain caractère

Explication : cette routine est une sous-routine de FORM_DO.

FORM_BUTTON(tree,obj,clicks,nouv_obj)

```
tree,obj,clicks : iexp
nouv_obj : ivar
```

Permet des entrées à l'aide de la souris dans un formulaire.

reponse	0 = un objet avec état EXIT a été cliqué en dernier lieu supérieur à 0 = le dialogue n'est pas encore terminé
tree	Adresse de l'arbre d'objets correspondant
obj	Objet actuel
clicks	Nombre de fois que les boutons de la souris ont été actionnés
nouv_obj	Nouvel objet actuel

Explication : cette routine est une sous-routine de FORM_DO.

4. Bibliothèque graphique (graphics library)

GRAF_RUBBERBOX(rx,ry,min_w,min_h,[last_w,last_h])

```
rx,ry,min_w,min_h : iexp
last_w,last_h : ivar
```

La fonction GRAF_RUBBERBOX affiche sur l'écran les contours d'un rectangle lorsqu'un bouton de la souris est actionné. Les coordonnées gauche x et supérieure y sont fixées mais le coin opposé en diagonale ainsi que la largeur et la hauteur du rectangle varient en fonction de la position du curseur de la souris. Il est conseillé de n'appeler la fonction que lorsque le bouton de la souris est enfoncé car la fonction est interrompue lorsqu'un bouton de la souris est relâché.

reponse	0 = Erreur apparue
rx	Coordonnée gauche x
ry	Coordonnée supérieure y
min_w	Largeur et hauteur
min_h	minimales du rectangle
last_w	Largeur et
last_h	hauteur du rectangle lors de l'arrêt de la fonction

Exemple :

```
DO
  ~EVNT_BUTTON(1,1,1,mx%,my%,bu%,kb%)
  ~GRAF_RUBBERBOX(mx%,my%,1,1,w%,h%)
  BOX mx%,my%,mx%+w%,my%+h%
LOOP UNTIL BTST(kb%,3)
```

--> Attend un clic de la souris. On peut alors étirer un rectangle (rubber) qui est ensuite dessiné. Arrêt en appuyant sur *Alternate* et le bouton gauche de la souris.

GRAF_DRAGBOX(iw,ih,ix,iy,rx,ry,rw,rh,[last_ix,last_iy])

```
iw,ih,ix,iy,rx,ry,rw,rh : iexp
last_ix,last_iy : ivar
```

Cette fonction sert à déplacer un rectangle (interne) à l'intérieur d'un rectangle plus grand et fixe (rectangle cadre). Il est conseillé de n'appeler la fonction que lorsque le bouton de la souris est enfoncé car la fonction est interrompue lorsqu'un bouton de la souris est relâché.

reponse	0 = Erreur apparue
iw	Largeur et
ih	hauteur du rectangle intérieur
ix	Coordonnées gauche x
iy	et supérieure y du rectangle intérieur lors de l'appel de la fonction
rx	Coordonnée gauche x
ry	Coordonnée supérieure y
rw	Largeur et
rh	hauteur du rectangle de cadre
last_ix	Coordonnées gauche x et
last_iy	et supérieure y du rectangle intérieur lors de l'arrêt de la fonction

GRAF_MOVEBOX(w,h,sx,sy,dx,dy)

```
w,h,sx,sy,dx,dy : iexp
```

La fonction GRAF_MOVEBOX dessine sur l'écran un rectangle mobile de largeur et hauteur fixes.

reponse	0 = Erreur apparue
w	Largeur et
h	hauteur du rectangle mobile
sx	Coordonnées gauche x et
sy	supérieure y du rectangle lors de l'appel de la fonction
dx	Coordonnées gauche x et
dy	supérieure y du rectangle lors de la fin de la fonction

Exemple :

```
~GRAF_MOVEBOX(10,20,200,20,80,120)
```

GRAF_GROWBOX(sx,sy,sw,sh,dx,dy,dw,dh)

```
sx,sy,sw,sh,dx,dy,dw,dh : iexp
```

Cette fonction dessine un rectangle en expansion.

reponse	0 = Erreur apparue
sx	Coordonnée gauche x
sy	Coordonnée supérieure y
sw	Largeur et
sh	hauteur du rectangle lors du début de la fonction
dx	Coordonnée gauche x
dy	Coordonnée supérieure y
dw	Largeur et
dh	hauteur du rectangle lors de la fin de la fonction

Exemple :

```
~GRAF_GROWBOX(100,100,10,10,0,0,300,180)
```

GRAF_SHRINKBOX(sx,sy,sw,sh,dx,dy,dw,dh)

```
sx,sy,sw,sh,dx,dy,dw,dh : iexp
```

Cette fonction dessine un rectangle en contraction.

reponse	0 = Erreur apparue
sx	Coordonnée gauche x
sy	Coordonnée supérieure y
sw	Largeur et
sh	hauteur du rectangle lors de la fin de la fonction
dx	Coordonnée gauche x
dy	Coordonnée supérieure y
dw	Largeur et
dh	hauteur du rectangle au début de la fonction

Exemple :

```
~GRAF_SHRINKBOX(0,0,300,180,100,100,10,10)
```

GRAF_WATCHBOX(tree,obj,in_state,out_state)

```
tree,obj,in_state,out_state : iexp
```

Cette fonction surveille un objet (BOX) dans un arbre Resource si le bouton de la souris est enfoncé. L'état de l'objet change chaque fois que le pointeur de la souris atteint ou quitte la zone correspondante (normal selected/normal).

reponse	Indique l'état du curseur de la souris lorsque le bouton de la souris est relâché, 0 = en dehors et 1 = à l'intérieur de l'objet à surveiller
tree	Adresse de l'arbre d'objets correspondant
obj	Numéro de l'objet à surveiller
in_state	Etat (OB_STATE) de l'objet à surveiller si le curseur de la souris figure à l'intérieur de l'objet
out_state	Etat (OB_STATE) de l'objet à surveiller si le curseur de la souris figure en dehors de l'objet. La définition des bits correspondante est indiquée sous OB_STATE.

GRAF_SLIDEBOX(tree,parent_obj,slider_obj,flag)

```
tree,parent_obj,slider_obj,flag : iexp
```

Cette fonction permet de tester ce qu'on appelle des *boutons-poussoirs*. Comme pour la fonction DRAG_BOX, les contours d'un rectangle (poussoir) peuvent ici être déplacés à l'intérieur d'un rectangle cadre. Le rectangle mobile ne peut toutefois être déplacé dans ce cas que verticalement ou horizontalement. D'autre part, le rectangle mobile doit encore être un objet-"enfant" du rectangle cadre à l'intérieur de l'arbre d'objets. Il est conseillé de n'appeler la fonction que lorsque le bouton de la souris est enfoncé car la fonction est interrompue lorsqu'un bouton de la souris est relâché.

reponse	Position relative du "bouton-poussoir" à l'intérieur du rectangle cadre dont il dépend : 0 = gauche ou haut 1000 = droite ou bas
tree	Adresse de l'arbre d'objets correspondant
parent_obj	Numéro d'objet du "retangle cadre"
slider_obj	Numéro d'objet du "bouton-poussoir"
flag	Orientation du bouton-poussoir 0 = horizontale 1 = verticale

GRAF_HANDLE(char_w,char_h,box_w,box_h)

```
char_w,char_h,box_w,box_h : ivar
```

Fournit la marque de la station de travail VDI, qui est utilisée sur un plan interne pour les appels AES, et l'extension d'un caractère dans le jeu de caractères standard.

reponse	Marque de la station de travail VDI actuelle
char_w	Largeur et
char_h	hauteur d'un caractère du jeu de caractères standard en points écran
box_w	Largeur et
box_h	hauteur d'un rectangle dans lequel s'inscrit un caractère standard

GRAF_MOUSE(forme_s,adr_motif)

```
forme_s,adr_motif : iexp
```

Cette fonction permet de sélectionner la forme du curseur de la souris parmi 8 formes prédéfinies ou une forme définie par l'utilisateur. Il est toutefois plus commode d'employer l'instruction DEFMOUSE à cet effet.

reponse 0 = Erreur apparue

forme_s Numéro de forme du curseur de la souris

0	= ARROW	Flèche
1	= TEXT_CRSR	Trait vertical
2	= HOURGLASS	Abeille
3	= POINT_HAND	Main avec index pointé
4	= FLAT_HAND	Main ouverte
5	= THIN_CROSS	Réticule mince
6	= THICK_CROSS	Réticule épais
7	= OUTL_CROSS	Contours de réticule
255	= USER_DEF	Défini par l'utilisateur
256	= M_OFF	Désactiver curseur de la souris
257	= M_ON	Activer curseur de la souris

adr_motif Adresse à partir de laquelle figurent les informations de bits pour la forme du curseur de la souris définie par l'utilisateur. 37 mots sont attendus avec les conventions suivantes :

```
1  = Coordonnées x et
2  = y du point d'action
3  = Nombre de niveaux de couleur, toujours 1
```

```
      4  = Couleur de masque, toujours 0
      5  = Couleur du curseur de la souris, toujours 1
6 à 21   =  Forme du masque
22 à 37 = Forme du curseur de la souris
```

GRAF_MKSTATE(mx,my,m_state,k_state)

```
mx,my,m_state,k_state : iexp
```

Cette fonction fournit les coordonnées actuelles du curseur de la souris ainsi que l'état des boutons de la souris et du clavier.

Il s'agit d'une routine AES de test de la souris. Contrairement à MOUSEX, etc..., le programme s'arrête lorsque la flèche se trouve dans ligne de menu.

reponse	Réservé, toujours 1
mx	Coordonnées actuelles x et
my	y du curseur de la souris
m_state	Etat des boutons de la souris
	Bit 0 = bouton gauche de la souris
	Bit 1 = bouton droit de la souris
k_state	Etat du clavier avec la définition des bits suivante :
	Bit mis = bouton enfoncé
	1 = touche Shift droite
	2 = touche Shift gauche
	4 = touche Control
	8 = touche Alternate

Bibliothèque Scrap (scrap library)

Contient des routines permettant l'échange de données entre différentes applications.

SCRP_READ(chemin$)

```
chemin$ : svar
```

Cette fonction lit le buffer de texte servant à la communication entre des programmes GEM exécutés successivement. Elle a été conçue avant tout pour permettre d'entreposer les noms de dossiers et fichiers.

reponse 0 = Erreur apparue
chemin$ Chemin d'accès pour le clip board.

SCRP_WRITE(chemin$)

```
chemin$ : sexp
```

Cette fonction écrit dans le scrap directory.

reponse 0 = Erreur apparue
chemin$ Nouveau chemin d'accès

Exemple :

```
-SCRP_WRITE("C:\TMP\A.X")
...
...
...
-SCRP_READ(a$)
PRINT a$
```

Bibliothèque de sélection de fichier (file selector library)

Permet de sélectionner un fichier dans le répertoire affiché ou bien directement en indiquant le chemin d'accès.

FSEL_INPUT(path$,name$,[button])

```
path$,name$ : svar
button : ivar
```

Cette fonction appelle le sélecteur de fichier standard. Elle équivaut à l'instruction *Fileselect* qui recompose toutefois automatiquement la désignation complète du fichier à partir des noms de chemin et de fichier alors que le programmeur peut ici le faire lui-même avec RINSTR(path$,"\").

reponse 0 = Erreur apparue
path$ Chemin d'accès présélectionné ; contient après arrêt de la fonction le chemin d'accès indiqué par l'utilisateur.
name$ Nom de fichier présélectionné ; contient après arrêt de la fonction le nom de fichier indiqué par l'utilisateur.

button Etat du bouton :
0 = bouton "Arrêt" sélectionné
1 = bouton "Ok" sélectionné

Bibliothèques de fenêtres (window library)

Se charge de la gestion des fenêtres.

WIND_CREATE(attr,wx,wy,ww,wh)

```
attr,wx,wy,ww,wh : iexp
```

Cette fonction déclare une nouvelle fenêtre et fixe les attributs et la taille maximale de la fenêtre. La fonction renvoie le code actuel de la fenêtre (window handle).

Cf. OPENW#n,x,y,w,h,attr

reponse 0 = Erreur apparue, sinon code de fenêtre

attr Attributs de fenêtres avec définition de bits suivante :

			N° de bit
&H0001	NAME	Barre de titre avec nom	0
&H0002	CLOSE	Champ de fermeture	1
&H0004	FULL	Champ pour taille maximale	2
&H0008	MOVE	Barre de déplacement	3
&H0010	INFO	Ligne d'informations	4
&H0020	SIZE	Réglage de la taille de la fenêtre	5
&H0040	UPARROW	Flèche vers le haut	6
&H0080	DNARROW	Flèche vers le bas	7
&H0100	VSLIDE	Bouton-poussoir vertical	8
&H0200	LFARROW	Flèche vers la gauche	9
&H0400	RTARROW	Flèche vers la droite	10
&H0800	HSLIDE	Bouton-poussoir horizontal	11

wx Coordonnée gauche x maximale
wy Coordonnée supérieure y
ww Largeur et
wh hauteur de la fenêtre

WIND_OPEN(handle,wx,wy,ww,wh)

```
handle,wx,wy,ww,wh : iexp
```

Cette fonction affiche sur l'écran une fenêtre préalablement créée avec WIND_CREATE.

Cf. OPENW

reponse	0 = Erreur apparue
handle	Code de la fenêtre
wx	Coordonnée gauche x
wy	Coordonnée supérieure y
ww	Largeur et
wh	hauteur de la fenêtre au début de la fonction

WIND_CLOSE(handle)

```
handle : iexp
```

Cette fonction représente la fonction inverse de WIND_OPEN et ferme la fenêtre voulue.

Cf. CLOSEW

reponse	0 = Erreur apparue
handle	Code de la fenêtre voulue

WIND_DELETE(handle)

```
handle : iexp
```

Cette fonction vide une fenêtre et libère à nouveau la place mémoire réservée ainsi que le code de fenêtre correspondant.

Cf. CLOSEW

reponse	0 = Erreur apparue
handle	Code de la fenêtre voulue

WIND_GET(handle,field,w1,w2,w3,w4)

```
handle,field : iexp
w1,w2,w3,w4 : ivar
```

Cette fonction fournit diverses informations sur une fenêtre, suivant le numéro de fonction.

reponse 0 = Erreur apparue
handle Code de la fenêtre voulue

field Suivant le numéro de fonction, diverses informations sur la fenêtre voulue sont renvoyées dans wi_gw1, wi_gw2, wi_gw3 et wi_gw4.

```
4   WX_WORKXYWH calcule les coordonnées de la zone de travail
    w1   Coordonnée gauche x
    w2   Coordonnée supérieure y
    w3   Largeur et
    w4   hauteur de la fenêtre voulue

5   WF_CURRXYWH calcule les coordonnées de la fenêtre entière y
    compris les bords
    w1   Coordonnée gauche x
    w2   Coordonnée supérieure y
    w3   Largeur et
    w4   hauteur de la fenêtre voulue

6   WF_PREVXYWH calcule la taille globale de la fenêtre précédente
    w1   Coordonnée gauche x
    w2   Coordonnée supérieure y
    w3   Largeur et
    w4   hauteur de la fenêtre voulue

7   WF_FULLXYWH calcule la taille globale de la fenêtre avec son
    extension maximale
    w1   Coordonnée gauche x
    w2   Coordonnée supérieure y
    w3   Largeur et
    w4   hauteur de la fenêtre voulue

8   WF_HSLIDE fournit la position du bouton-poussoir horizontal
    w1   (1 = tout à gauche, 1000 = tout à droite)

9   WF_VSLIDE fournit la position du bouton-poussoir vertical
    w1   (1 = tout en haut, 1000 = tout en bas)
```

```
10  WF_TOP fournit le code de la fenêtre du haut
    w1    Code de la fenêtre

11  WF_FIRSTXYWH fournit les coordonnées du premier rectangle à
    l'intérieur de la liste de rectangles de la fenêtre voulue
    w1    Coordonnée gauche x
    w2    Coordonnée supérieure y
    w3    Largeur et
    w4    hauteur de la fenêtre voulue

12  WF_NEXTXYWH fournit les coordonnées du prochain rectangle à
    l'intérieur de la liste de rectangles de la fenêtre voulue
    w1    Coordonnée gauche x
    w2    Coordonnée supérieure y
    w3    Largeur et
    w4    hauteur de la fenêtre voulue

13  WF_RESVD Réservé

15  WF_HSLIZE calcule la taille du bouton-poussoir par rapport
    à la barre :
    w1    -1 = Taille minimale
               (1 = petite, 1000 = toute la largeur)

16  WF_VSLIZE calcule la taille du bouton-poussoir vertical par
    rapport à la barre :
    w1    -1 = Taille minimale
               (1 = petite, 1000 = toute la largeur)
```

WIND_SET(handle,field,w1,w2,w3,w4)

```
handle,field : iexp
w1,w2,w3,w4 : ivar
```

Cette fonction modifie différentes parties de la fenêtre voulue suivant le numéro de fonction.

reponse 0 = Erreur apparue
handle Code de la fenêtre voulue

field Différentes parties de la fenêtre voulue sont modifiées suivant le numéro de fonction

```
1   WF_KIND   fixe de nouvelles parties de fenêtre
              (comme défini sous WIND_CREATE)
    w1    Nouvelle partie de fenêtre
```

```
2   WF_NAME   fixe un nouveau titre de fenêtre
    w1    et
    w2    contiennent l'adresse de la chaine

3   WF_INFO   fixe une nouvelle ligne d'informations
    w1    et
    w2    contiennent l'adresse de la chaine

5   WF_CURRXYWH fixe la taille de la fenêtre
    w1    Coordonnée gauche x
    w2    Coordonnée supérieure y
    w3    Largeur et
    w4    hauteur de la fenêtre voulue

8   WF_HSLIDE positionne le bouton-poussoir horizontal
    w1    (1 = tout à gauche, 1000 = tout à droite)

9   WF_VSLIDE positionne le bouton-poussoir vertical
    w1    (1 = tout en haut, 1000 = tout en bas)

10  WF_TOP fixe la fenêtre du haut (actuelle)

14  WF_NEWDESK fixe un nouvel arbre de menus du bureau
    w1    Octet fort
    w2    Octet faible de l'adresse de l'arbre d'objets
    w3    Numéro du premier objet à dessiner

15  WF_HSLIZE fixe la taille du bouton-poussoir horizontal par
    rapport à la barre
    w1    -1 = Taille minimale
               (1 = petite, 1000 = toute la largeur)

16  WF_VSLIZE fixe la taille du bouton-poussoir vertical par
    rapport à la barre
    w1    -1 = Taille minimale
               (1 = petite, 1000 = toute la largeur)
```

WIND_FIND(fx,fy)

```
fx,fy : iexp
```

Cette fonction détermine le code d'une fenêtre située sur les coordonnées spécifiées.

reponse 0 = aucune fenêtre ne correspond à ces coordonnées, sinon code de la fenêtre trouvée
fx Coordonnées x et
fy y de l'emplacement de l'écran correspondant

WIND_UPDATE(flag)

```
flag : iexp
```

Cette fonction coordonne toutes les dispositions relatives à la construction de l'écran, notamment en ce qui concerne les menus déroulants.

reponse 0 = Erreur apparue
flag Numéro de fonction

0 = END_UPDATE	Construction de l'écran achevée
1 = BEG_UPDATE	Construction de l'écran débute
2 = END_MCTRL	L'application abandonne le contrôle de la souris
3 = BEG_MCTRL	L'application prend totalement en charge le contrôle de la souris, notamment en verrouillant tous les menus déroulants.

WIND_CALC(w_type,attr,ix,iy,iw,ih,ox,oy,ow,oh)

```
w_type,attr,ix,iy,iw,ih : iexp
ox,oy,ow,oh : ivar
```

Cette fonction calcule les dimensions globales d'une fenêtre à partir de la taille de la zone de travail ou inversement la zone de travail à partir de la taille de la fenêtre.

reponse 0 = Erreur apparue
w_type 0 = Calculer dimensions globales
1 = Calculer zone de travail

attr Eléments de fenêtre :

			N° de bit
&H0001	NAME	Barre de titre avec nom	0
&H0002	CLOSE	Champ de fermeture	1
&H0004	FULL	Champ pour taille maximale	2
&H0008	MOVE	Barre de déplacement	3
&H0010	INFO	Ligne d'informations	4
&H0020	SIZE	Réglage de la taille de la fenêtre	5
&H0040	UPARROW	Flèche vers le haut	6
&H0080	DNARROW	Flèche vers le bas	7
&H0100	VSLIDE	Bouton-poussoir vertical	8
&H0200	LFARROW	Flèche vers la gauche	9
&H0400	RTARROW	Flèche vers la droite	10
&H0800	HSLIDE	Bouton-poussoir horizontal	11

ix
iy
iw
ih Coordonnées connues

ox
oy
ow
oh Coordonnées calculées

Bibliothèque Resource (resource library)

Contient des routines permettant de réaliser une interface utilisateur graphique simplifiant l'échange de données entre l'utilisateur et le programme indépendamment de la résolution sélectionnée.

RSRC_LOAD(nom$)

```
nom$ : sexp
```

Cette fonction réserve de la place mémoire et charge un fichier Resource. Des pointeurs internes sont ensuite fixés et des coordonnées sont converties du format de caractère en format de points écran. Pour les Resources définis par l'utilisateur lui-même dans la mémoire, RSRC_OBFIX doit encore être utilisé à cet effet.

reponse 0 = Erreur apparue
nom$ Chemin d'accès du fichier Resource voulu

Exemple :

```
-RSRC_LOAD("TEST.RSC")
```

RSRC_FREE()

Cette fonction libère à nouveau la place mémoire réservée avec RSRC_LOAD.

reponse 0 = Erreur apparue

Exemple :

```
-RSRC_FREE()
```

RSRC_GADDR(type,index,adr)

```
type,index : iexp
adr : ivar
```

Cette fonction détermine l'adresse d'une structure Resource après chargement de celle-ci avec RSRC_LOAD. Cette fonction n'est normalement utilisée que pour les arbres et les alertes (ad_frstr).

reponse 0 = Erreur apparue

type Type de structure recherché

0	Arbre d'objets
1	OBJECT
2	TEDINFO
3	ICONBLK
4	BITBLK
5	STRING
6	imagedata
7	obspec
8	te_ptext
9	te_ptmplt
10	te_pvalid
11	ib_pmask
12	ib_pdata
13	ib_ptext
14	bi_pdata
15	ad_frstr
16	ad_frimg

index Numéro d'objet de la structure recherchée
adr Adresse de la structure recherchée

Exemple :

```
~RSRC_GADDR(0,0,TREE%)
```

RSRC_SADDR(type,index,adr)

```
type,index : iexp
adr : ivar
```

Cette fonction fixe des adresses dans des Resources.

reponse 0 = Erreur apparue

type

0	Arbre d'objets
1	OBJECT
2	TEDINFO
3	ICONBLK
4	BITBLK
5	STRING
6	imagedata
7	obspec
8	te_ptext
9	te_ptmplt
10	te_pvalid
11	ib_pmask
12	ib_pdata
13	ib_ptext
14	bi_pdata
15	ad_frstr
16	ad_frimg

index Position à l'intérieur de la structure de données dans laquelle re_saddr est écrit.
adr L'adresse à sauvegarder.

RSRC_OBFIX(tree,obj)

```
tree,obj : iexp
```

Cette fonction convertit les coordonnées d'objet calculées en caractères en coordonnées calculées en points écran. Cela est nécessaire si des objets n'ont pas été chargés avec RSRC_LOAD mais ont été directement intégrés dans le programme.

reponse Réservé, toujours 1
tree Adresse de l'arbre d'objets correspondant
obj Numéro de l'objet à convertir

Bibliothèque Shell (shelbind.lib)

Regroupe des routines permettant de faire appeler une application par un autre programme, l'application et son environnement étant conservés.

SHEL_READ(cmd_string$,tail_string$)

```
cmd_string$,tail_string$ : svar
```

Cette fonction fournit le nom et la ligne d'instruction avec laquelle l'application a été appelée à partir du bureau (lignes d'instruction déclarées par l'application).

reponse	0 = Erreur apparue
cmd_string$	Adresse de la chaîne pour la ligne d'instruction
tail_string$	Adresse de la chaîne pour le nom

SHEL_WRITE(prg,grf,gem,cmd$,num$)

```
prg,grf,gem : iexp
cmd$,num$ : sexp
```

Cette fonction indique à l'AES qu'une autre application doit être lancée. Contrairement à ce qui se passe avec l'instruction GEMDOS pexec, le programme actuel ne reste cependant pas résident en mémoire.

reponse	0 = Erreur apparue
prg	0 = Retour au bureau
	1 = Charger nouveau programme
grf	0 = Programme TOS
	1 = Application graphique
gem	0 = Pas une application GEM
	1 = Application GEM
cmd$	Adresse de la chaîne avec la ligne d'instruction
num$	Adresse de la chaîne avec le nom

Exemple :

```
~SHEL_WRITE(1,1,1,"","GFABASIC.PRG")
```

--> Cela signifie que le GFA-BASIC sera relancé après abandon du BASIC et retour au bureau.

SHEL_GET(nmb,x$)

```
nmb : iexp
x$ : svar
```

Cette fonction permet de lire dans la mémoire d'environnement.

reponse	0 = Erreur apparue
nmb	Nombre d'octets à lire
x$	Chaîne de caractères pour recevoir les caractères lus

SHEL_PUT(len,x$)

```
len : iexp
x$ : sexp
```

Cette fonction permet d'écrire un nombre de caractères à spécifier dans la mémoire d'environnement de GEM (desktop.inf).

reponse	0 = Erreur apparue
x$	Chaîne de caractères contenant les caractères à écrire
len	Nombre d'octets à écrire

Exemple :

```
' Déclarer GFA-BASIC
~SHEL_GET(2000,a$)
q%=INSTR(a$,CHR$(26))
IF q%
  a$=LEFT$(a$,q%-1)
  IF INSTR(a$,GFABASIC.PRG")=0
    a$=a$+"#G 03 04   c:\GFA\GFABASIC.PRG@ *.GFA@ "+MKI$(&hd0a)+CHR$(26)
  ~SHEL_PUT(LEN(a$),a$)
  ENDIF
ENDIF
```

--> Le programme déclare le C:\GFA\GFABASIC.PRG de sorte que ce GFA-BASIC sera lancé à la suite de chaque double clic sur des fichiers comportant l'extension .GFA, dans le cas bien sûr où vous possédez un disque dur.

Remarque : sauvegarde du travail avec :

```
-SHEL_GET(3000,a$)
OPEN"O",#1,"C:\DESKTOP.INF"
PRINT #1,LEFT$(A$,INSTR(A$,CHR$(26))
CLOSE #1
' Le CHR$(26) est très important
```

SHEL_FIND(chemin$)

```
chemin$ : svar
```

Cette fonction recherche un fichier dans le dossier actuel, dans le répertoire actuel du lecteur activé et sur le lecteur "A:". En cas de succès, le chemin complet de ce fichier est sorti (xx.x ou \xx.x ou a:\xx.x).

reponse 0 = Erreur apparue
chemin$ Chaîne contenant le nom du fichier recherché. Après la fin de la fonction, cette chaîne contient le chemin d'accès au fichier complet. C'est pourquoi la longueur maximale doit être spécifiée.

SHEL_ENVRN(adr,rech$)

```
adr : avar (32 bits)
rech$ : sexp
```

Cette fonction sert à obtenir la valeur de données variables dans l'environnement du DOS (par exemple PATH=A:\).

reponse Réservé, toujours 1
adr Adresse avant laquelle figure la chaîne spécifiée dans rech$.
rech$ Chaîne contenant le critère de recherche (par exemple A:\)

Exemple :

```
PRINT SHEL_ENVRN(a%,"P")
PRINT CHAR{a%-1}
```

--> Affichage : PATH=;A:\

PROGRAMMES D'EXEMPLE

Cette dernière section vous propose quatre programmes d'exemple à propos de la bibliothèque graphique, des sélecteurs d'objet, des barres de menus et de la programmation des fenêtres.

```
' Bibliothèque graphique
'
REPEAT
  CLS
  PRINT "rubber = 1, drag = 2, move = 3, grow_shrink = 4, fin = 0"
  INPUT "choix : ";choix%
  ON choix% GOSUB rubber,drag,move,grow_shrink
UNTIL choix%=0
'
EDIT
'
PROCEDURE rubber
  GRAPHMODE 3
  DEFFILL 1,2,4
  REPEAT
    MOUSE mx%,my%,mk%
    IF mk% AND 1
      x1%=mx%
      y1%=my%
      ~GRAF_RUBBERBOX(x1%,y1%,16,16,lx%,ly%)
      PBOX x1%,y1%,x1%+lx%,y1%+ly%
    ENDIF
  UNTIL mk% AND 2
RETURN
'
PROCEDURE drag
  GRAPHMODE 1
  DEFFILL 1,2,4
  REPEAT
    MOUSE mx%,my%,mk%
    BOX 50,50,200,150
    BOX 40,40,400,300
    IF mk% AND 1
      ~GRAF_DRAGBOX(150,100,50,50,40,40,400,300,lx%,ly%)
      PBOX lx%,ly%,lx%+150,ly%+100
    ENDIF
  UNTIL mk% AND 2
RETURN
```

```
'
PROCEDURE move
  GRAPHMODE 1
  DEFFILL 1,2,4
  b%=64
  h%=64
  FOR i%=0 TO 639-b% STEP b%
    FOR j%=0 TO 399-h% STEP h%
      ~GRAF_MOVEBOX(b%,h%,i%,j%,639-i%,399-j%)
    NEXT j%
  NEXT i%
RETURN
'
PROCEDURE grow_shrink
  GRAPHMODE 1
  ~GRAF_GROWBOX(319,199,16,16,0,0,639,399)
  PRINT "C'était graf_growbox"
  PRINT "Veuillez frapper une touche"
  ~INP(2)
  ~GRAF_SHRINKBOX(319,199,16,16,0,0,639,399)
  PRINT "C'était graf_shrinkbox"
  PRINT "Veuillez frapper une touche"
  ~INP(2)
RETURN
```

--> Demande la routine voulue et saute à la procédure correspondante. Si un 1 est spécifié, les contours d'un rectangle apparaissent sur l'écran si le bouton gauche de la souris est enfoncé. Les coordonnées gauche x et supérieure y sont fixes mais le coin opposé suivant la diagonale ainsi que la largeur et la hauteur du rectangle varient en fonction de la position actuelle du curseur.

Lorsque le bouton de la souris est relâché, un rectangle plein est dessiné sur les coordonnées du dernier rectangle affiché. La procédure est abandonnée lorsque vous appuyez sur le bouton droit de la souris.

Si un 2 est spécifié, une grande boîte cadre est dessinée dans laquelle s'inscrit un rectangle plus petit. En tenant le bouton gauche de la souris enfoncé, vous pouvez déplacer le rectangle intérieur à l'intérieur du rectangle cadre. Lorsque le bouton de la souris est relâché, un rectangle plein est dessiné sur les coordonnées du dernier rectangle intérieur affiché. La procédure est abandonnée lorsque vous appuyez sur le bouton droit de la souris.

Si vous choisissez 3, on saute à une procédure qui appelle GRAF_MOVEBOX pour afficher une série de rectangles se mouvant de façon circulaire autour du centre de l'écran.

Si vous indiquez 4, enfin, un rectangle en expansion sera dessiné, on attendra qu'une touche soit frappée et un rectangle en contraction sera finalement affiché.

Le programme s'arrête dès que vous spécifiez 0.

```
' Gestion de boite de dialogue
'
DIM r%(3)
'          (* Indices de Resource pour DIALOG *)
form1%=0     !Formulaire/dialogue *)
icone1%=1    !ICON dans l'arbre FORM1 *)
pre%=2       !FTEXT dans l'arbre FORM1 *)
nom%=3       !FTEXT dans l'arbre FORM1 *)
rue%=4       !FTEXT dans l'arbre FORM1 *)
vil%=5       !FTEXT dans l'arbre FORM1 *)
arret%=6     !BUTTON dans l'arbre FORM1 *)
ok%=7        !BUTTON dans l'arbre FORM1 *)
r%(1)=8      !BUTTON dans l'arbre FORM1 *)
r%(2)=9      !BUTTON dans l'arbre FORM1 *)
r%(3)=10     !BUTTON dans l'arbre FORM1 *)
sortie%=11   !STRING dans l'arbre FORM1 *)
'
-RSRC_FREE()
RESERVE FRE(0)+32000
charger_ok!=RSRC_LOAD("\dialog.rsc") !Charger Resource
IF NOT charger_ok!
  ALERT 1,"RSC non trouvé",1," Return ",a%
  RESERVE FRE(0)+32000
  EDIT
ENDIF
-RSRC_GADDR(0,0,tree%)              !Déterminer adresse de l'arbre d'objets
-FORM_CENTER(tree%,x%,y%,w%,h%)     !Centrer les coordonnées de l'arbre d'objets
'
' Prédéfinir les textes dans les champs Edit
CHAR{{OB_SPEC(tree%,pre%)}}="Hector"
CHAR{{OB_SPEC(tree%,nom%)}}="Berlioz"
CHAR{{OB_SPEC(tree%,rue%)}}="77, rue Fantastique"
CHAR{{OB_SPEC(tree%,vil%)}}="Lyon"
'
-OBJC_DRAW(tree%,0,1,x%,y%,w%,h%)  !Dessiner l'arbre d'objets
'
```

```
REPEAT
  ex%=FORM_DO(tree%,0)      !Objet cliqué avec état Exit ?
  '
  ' Charger textes des champs Edit dans les chaines appropriées
  prenom$=CHAR{{OB_SPEC(tree%,pre%)}}
  nom$=CHAR{{OB_SPEC(tree%,nom%)}}
  rue$=CHAR{{OB_SPEC(tree%,rue%)}}
  ville$=CHAR{{OB_SPEC(tree%,vil%)}}
  '
  FOR i%=1 TO 3
    IF BTST(OB_STATE(tree%,r%(i%)),0)   !Quel bouton radio
      radio%=r%(i%)                     !a été cliqué ?
    ENDIF
  NEXT i%
UNTIL ex%=ok% OR ex%=arret%
'
~RSRC_FREE()   !Libérer à nouveau la place mémoire réservée
RESERVE FRE(0)+32000
'
CLS
PRINT "Fin avec : ";ex%
PRINT "Prénom   : ";prenom$
PRINT "Nom      : ";nom$
PRINT "Rue      : ";rue$
PRINT "Ville    : ";ville$
PRINT "Radio    : ";radio%
```

--> Le fichier Resource "DIALOG.RSC" est chargé, l'adresse de départ de l'arbre d'objets est calculée et les coordonnées d'objet sont centrées. Les chaînes par défaut sont ensuite écrites dans les champs Edit. Cela est effectué à l'aide de la fonction OB_SPEC.

Cette fonction fournit un pointeur sur une structure TEDINFO contenant différentes informations sur un texte, par exemple l'adresse du texte et des modèles correspondants, sa longueur, etc... Cette structure TEDINFO contient à son tour un pointeur sur la chaîne elle-même. Le chemin conduisant de l'objet à la chaîne se présente donc comme suit :

Objet --> Structure TEDINFO --> Chaîne

OB_SPEC(tree%,pre%) fournit un pointeur sur une structure TEDINFO.

{OB_SPEC(tree%,pre%)} fournit le pointeur désigné par la structure TEDINFO.

CHAR{{OB_SPEC(tree%,pre%)}} fournit la chaine de caractères définie dans le Resource Construction Set, en l'occurrence : 01234567890123456789O1

CHAR{{OB_SPEC(tree%,pre%)}}="Hector" affecte une nouvelle valeur (Hector en l'occurrence) à cette chaine de caractères prédéfinie.

L'arbre d'objets est ensuite dessiné et examiné à l'intérieur de la boucle REPEAT-UNTIL à l'aide de la routine FORM_DO. On teste lors de chaque parcours si un objet possédant un état Exit a été cliqué, et si oui, la boucle est abandonnée. C'est la comparaison de la valeur de réponse de FORM_DO avec les numéros d'objet des boutons Arrêt et Ok qui sert de critère d'arrêt.

A l'intérieur de la boucle FOR-NEXT, BTST est utilisé pour examiner l'état d'objet des boutons radio. A cet effet, les numéros d'objet des boutons radio ont été, pour simplifier, chargés au début du programme dans un tableau entier (r%(1) à r%(3)), ce qui permet un test très simple dans le cadre de la boucle FOR-NEXT. OB_STATE fournit l'état d'objet (voyez aussi au début de ce chapitre) et BTST teste si le bit zéro du bouton radio correspondant est fixé ou non (c'est-à-dire si l'objet a été appelé ou non). Comme il n'est jamais possible d'appeler simultanément plus d'un bouton radio du même groupe, le numéro d'objet du bouton radio correspondant est placé dans une variable de sorte que si le test est interrompu c'est le numéro d'objet du dernier bouton radio sélectionné qui figure dans cette variable.

RSRC_FREE libère enfin à nouveau la place mémoire réservée pour le Resource (très important !, sinon la place mémoire disponible se restreindra lors de chaque lancement du programme, ce qui conduira bientôt au plantage de la machine).

Pour terminer est indiqué le numéro de l'objet avec lequel la routine FORM_DO a été abandonnée (6 = Arrêt ou 7 = Ok). Viennent ensuite les chaînes de caractères qui figuraient dans les 4 champs Edit lorsque la routine a été abandonnée, puis le numéro d'objet du dernier bouton radio sélectionné.

```
' Programmation de barre des menus
'
' Réserver place mémoire pour Resource, charger Resource
' et afficher barre des menus.
RESERVE FRE(0)-33000
IF RSRC_LOAD("\menuel.rsc")=0
  ALERT 1,"Fichier Resource|non trouvé.",1," Return ",a%
  RESERVE FRE(0)+33000
  EDIT
ENDIF
~RSRC_GADDR(0,0,adr_menu%)
~MENU_BAR(adr_menu%,1)
'
' Mettre en place buffer de message et préparer des variables utiles
```

```
DIM buffer_message%(3)
adr_mes%=V:buffer_message%(0)
ABSOLUTE type_mes&,adr_mes%
ABSOLUTE titre_m&,adr_mes%+6
ABSOLUTE entree_m&,adr_mes%+8
'
REPEAT
  ~EVNT_MULTI(&X110000,0,0,0,0,0,0,0,0,0,0,0,0,0,adr_mes%,100)
  ' si un point du menu a été sélectionné
  IF type_mes&=10
    ' si une entrée autre que la dernière fois a été sélectionnée
    IF no_obj%<>entree_m&
      no_obj%=entree_m&
      titre$=CHAR{OB_SPEC(adr_menu%,titre_m&)}
      entree$=CHAR{OB_SPEC(adr_menu%,entree_m&)}
      PRINT AT(3,20);"Titre du menu : ";titre$;SPC(15)
      PRINT AT(3,22);"Entrée du menu : ";entree$;SPC(15)
      ~MENU_TNORMAL(adr_menu%,titre_m&,1)
    ENDIF
  ENDIF
UNTIL MOUSEK=2
'
' Supprimer barre des menus, éliminer Resource de la mémoire et
' restaurer la mémoire réservée
~MENU_BAR(adr_menu%,0)
~RSRC_FREE()
RESERVE FRE(0)+33000
END
```

--> Au début du programme, 33 kilooctets de place mémoire sont réservés pour un fichier Resource (un fichier de ce type ne peut avoir une taille supérieure à 33 Ko). Cette place mémoire est restituée à l'intérpréteur GFA-BASIC à la fin du programme.

Une fois la place mémoire réservée, le fichier Resource est chargé, l'adresse de l'arbre de menus est déterminée et le menu est affiché sur l'écran.

La gestion du menu s'opère avec une fonction EVNT_MULTI qui surveille l'arbre de menus. Si des éléments de l'arbre de menus sont appelés par l'utilisateur, des messages sont écrits dans ce qu'on appelle un buffer de message. Il s'agit d'une zone de mémoire d'une longueur de 16 octets organisée en 8 mots (= 8 fois 2 octets). Ce buffer est mis en place dans un tableau créé avec DIM.

Les deux premiers octets de ce buffer contiennent un 10 si une entrée du menu a été appelée. Dans ce cas, le numéro d'objet du titre de menu sous lequel une entrée a été sélectionné est inscrit à partir du sixième octet, le numéro d'objet de l'entrée figurant à partir du huitième octet du buffer de message.

Les autres éléments du buffer ne jouent aucun rôle pour le moment. Pour que les éléments significatifs puissent être appelés plus aisément, des noms de variables leur sont affectés avec ABSOLUTE.

Vient ensuite une boucle qui peut être abandonnée en appuyant sur le bouton droit de la souris. C'est là que figure la fonction EVNT_MULTI qui surveille le menu. Si une entrée du menu a été sélectionnée, les numéros d'objet du titre du menu déroulé et de l'entrée de menu sélectionnée figurent dans les variables titre_m& et entree_m&. A l'aide de ces deux variables, les fonctions CHAR et OB_SPEC permettent de déterminer les textes du titre du menu et de l'entrée du menu. Il faut savoir à ce propos que OB_SPEC constitue un pointeur sur une structure TEDINFO dont les quatre premiers octets constituent eux-mêmes un pointeur sur le texte de l'objet.

OB_SPEC(...) est donc l'adresse de la chaîne concernée, qui peut être lue avec CHAR puisqu'elle se termine par un octet nul (voyez aussi le programme de gestion d'un sélecteur d'objet).

Une fois les deux textes sortis sur l'écran, le titre du menu inversé repasse en état non inversé.

```
' Programmation des fenêtres
'
DEFFILL 1,2,4
PBOX 0,19,639,399
DEFFILL 1,0
'
DIM buffer_message%(3)  ! 16 octets
adr_mes%=V:buffer_message%(0)
'
ABSOLUTE word0&,adr_mes%
ABSOLUTE x&,adr_mes%+8
ABSOLUTE y&,adr_mes%+10
ABSOLUTE w&,adr_mes%+12
ABSOLUTE h&,adr_mes%+14
'
handle&=WIND_CREATE(&X101111,0,19,639,380)
'
titre$="Window"
adr_tit%=V:titre$
~WIND_SET(handle&,2,CARD(SWAP(adr_tit%)),CARD(adr_tit%),0,0)
~WIND_OPEN(handle&,100,100,200,100)
~WIND_GET(handle&,4,wx&,wy&,ww&,wh&)
PBOX wx&,wy&,wx&+ww&,wy&+wh&
'
dehors!=FALSE
```

```
REPEAT
  ~EVNT_MULTI(&X110000,0,0,0,0,0,0,0,0,0,0,0,0,0,adr_mes%,100)
  SELECT word0&
  CASE 22                              ! WM_CLOSED
    dehors!=TRUE
  CASE 23                              ! WM_FULLED
    ~WIND_SET(handle&,5,1,19,638,380)
    ~WIND_GET(handle&,4,wx&,wy&,ww&,wh&)
    PBOX wx&,wy&,wx&+ww&,wy&+wh&
    word0&=0
  CASE 27,28                           ! WM_SIZED, WM_MOVED
    IF w&<100
      w&=100
    ENDIF
    IF h&<80
      h&=80
    ENDIF
    ~WIND_SET(handle&,5,x&,y&,w&,h&)
    ~WIND_GET(handle&,4,wx&,wy&,ww&,wh&)
    PBOX wx&,wy&,wx&+ww&,wy&+wh&
    word0&=0
  ENDSELECT
UNTIL dehors!
~WIND_CLOSE(handle&)
~WIND_DELETE(handle&)
```

--> Au début du programme, un buffer de message est mis en place pour recevoir les messages sur les éléments des bords de fenêtre appelés. Des noms de variables sont affectés avec ABSOLUTE aux éléments de cette zone de mémoire qui jouent un rôle dans ce programme.

La fenêtre est ensuite mise en place avec WIND_CREATE et un nom lui est affecté avec WIND_SET. Une fois la fenêtre ouverte avec WIND_OPEN, la zone de travail de la fenêtre est déterminée puis effacée avec un rectangle blanc.

Dans la boucle qui suit, la sélection des éléments de manipulation de la fenêtre est surveillée par une fonction EVNT_MULTI. Les différents éléments sont traités à la suite des instructions CASE correspondantes. Si l'élément de fermeture de la fenêtre a été sélectionné, le programme est terminé après que la fenêtre ait été refermée et éliminée de la mémoire. Vous trouverez ce programme sur votre disquette Gfa Basic, sous le nom de WINDOW.LST.

12. ANNEXES

COMPATIBILITE AVEC GFA-BASIC 2.xx

Il est possible d'utiliser sous GFA-BASIC 3.0 les programmes réalisés sous des versions plus anciennes de GFA-BASIC. A cet effet, ces programmes doivent être sauvegardés sous l'ancienne version comme des fichiers ASCII (Save,A) puis rechargés sous la nouvelle version avec Merge. Une fois ce transfert opéré, ces programmes peuvent alors être traités normalement sous GFA-BASIC 3.0 avec Save et Load.

Toutes les instructions contenues dans les anciennes versions de l'interpréteur figurent également dans la version 3.0. Il y a cependant quelques petites différences dans la signification des instructions qui rendent une adaptation nécessaire.

MUL DIV

Les instructions MUL et DIV de la version 3.0 ne traitent réellement que des paramètres entiers lorsqu'elles sont employées avec des variables entières (|,&,%).

Sous les versions antérieures :

```
a%=10
MUL a%,2.5
```

aboutissait à ce que a% reçoive la valeur 25. La version 3.0 ignore désormais les chiffres après la virgule pour les paramètres de l'instruction MUL, de sorte que a% vaudrait ici finalement 20. Cette incompatibilité était le prix à payer pour l'emploi d'une véritable arithmétique entière qui rend ce groupe d'instructions beaucoup plus rapides que dans les versions antérieures de GFA-BASIC. Cela ne concerne toutefois que les variables entières.

PRINT USING

En cas de débordement, PRINT USING ne se contente plus de sortir le nombre non formaté mais sort uniquement les chiffres trouvant place dans le format voulu. Le format sorti sera donc toujours le bon mais avec dans certains cas des valeurs erronées. D'autre part, avec le format exponentiel, même un nombre élevé de caractères "^" seront toujours traités correctement et l'exposant sera adapté s'il y a plusieurs chiffres avant la virgule.

CLS - PRINT TAB

CLS entraîne maintenant la sortie d'un ESC-E-CR de sorte que le premier PRINT TAB est correctement exécuté.

KEYPAD

Un programme qui teste les touches du bloc numérique avec *Alternate* et/ou *Control* doit maintenant employer un KEYPAD 0 pour désactiver la définition spéciale du clavier. Il en va de même pour les touches de fonction avec *Alternate*.

MOUSEX - MOUSEY

Lorsque des fenêtres sont activées et qu'elles sont testées avec MOUSEX/MOUSEY, des coordonnées négatives sont obtenues pour les valeurs situées au-dessus ou à gauche des limites de la fenêtre (ou de CLIP OFFSET). Sous la version 2.X, CARD(MOUSEX) ou CAR(MOUSEY) était renvoyé.

Les programmes réalisés sous l'interpréteur peuvent ensuite être considérablement accélérés en utilisant le compilateur GFA-BASIC 3.0. Ce compilateur permet ainsi de réaliser des programmes qui peuvent tourner sans l'interpréteur. Il est par ailleurs également possible d'utiliser l'interpréteur RUN-ONLY qui est fourni avec GFA-BASIC 3.0. Les acheteurs d'un GFA-BASIC dûment enregistrés peuvent fournir cet interpréteur RUN-ONLY avec les programmes qu'ils diffusent, cet interpréteur permettant à des personnes n'ayant pas acheté l'interpréteur GFA-BASIC d'utiliser malgré tout des programmes GFA-BASIC.

Tables GEMDOS

~GEMDOS(0) **ptermo()**

Termine un programme. Ne doit pas être utilisé sous GFA-BASIC.

r%=GEMDOS(1) **cconin()**

Lire un caractère au clavier avec affichage sur l'écran (cf. INP(2)).

```
r% Bits 0 à 7 : code ASCII, 16 à 23 : code clavier, 24 à 31 : touches de
commutation du clavier.
```

~GEMDOS(2,z%) **cconout()**

Sortir un caractère sur l'écran (cf. OUT 2,z%).

```
z% Bits 0 à 7 : code ASCII du caractère.
```

r%=GEMDOS(3) **cauxin()**

Lit un caractère sur l'interface sérielle (cf. INP(1)).

```
r% Code du caractère lu.
```

~GEMDOS(4,z%) **cauxout()**

Sortir un caractère sur l'interface sérielle (cf. OUT 1,z%).

```
z% Code du caractère à sortir.
```

~GEMDOS(5,z%) **cprnout()**

Sortir un caractère sur l'interface imprimante (cf. OUT 0,z%).

```
z% Code du caractère à sortir.
```

r%=GEMDOS(6,z%) **crawio()**

Lire ou écrire un caractère au clavier sans affichage sur l'écran (cf. OUT 2,z% ainsi que INKEY$).

```
r% Si z%=255, le caractère lu au clavier.
z% Code du caractère à sortir, si z%=255, lire un caractère.
```

r%=GEMDOS(7) **crawcin()**

Lire un caractère au clavier sans affichage sur l'écran (cf. INP(2)).

```
r% Code du caractère lu.
```

r%=GEMDOS(8) **cnecin()**

Comme GEMDOS(7) mais sans tenir compte des caractères de commande, par exemple CTRL+C.

```
r% Code du caractère lu.
```

~GEMDOS(9,L:adr%) **cconws()**

Sort une chaîne de caractères sur l'écran.

```
adr% Adresse de la chaine de caractères qui doit se terminer par un octet nul.
```

~GEMDOS(10,L:adr%) **cconrs()**

Lit une chaîne de caractères au clavier (CTRL+C entraîne un plantage).

```
adr%   Buffer pour la chaine de caractères : le premier octet spécifie le nombre
       de caractères à lire, le second octet indique le nombre d'octets lus, puis
       vient la chaine.
```

r!=GEMDOS(11) **cconis()**

Examine s'il y a un caractère dans le buffer clavier.

```
r! est TRUE s'il y a un caractère, sinon FALSE.
```

~GEMDOS(14,d%) **dsetdrv()**

Fixe le lecteur actuel ; 0 = A, 1 = B, etc... (cf. CHDRIVE).

```
d% Numéro du lecteur.
```

r!=GEMDOS(16) **cconos()**

Examine si un caractère peut être sorti sur l'écran.

```
r! Etat de sortie, toujours TRUE.
```

r!=GEMDOS(17) **cprnos()**

Examine si la sortie est possible sur l'interface parallèle.

```
r! Etat de sortie, TRUE si prêt à recevoir (il s'agit normalement de l'imprimante).
```

r!=GEMDOS(18) **cauxis()**

Examine si un caractère est présent sur l'interface sérielle.

```
r! TRUE si un caractère est présent, sinon FALSE.
```

r!=GEMDOS(19) **cauxos()**

Examine l'état de sortie de l'interface sérielle.

```
r! TRUE si un caractère peut être sorti, sinon FALSE.
```

r%=GEMDOS(25) **dgetdrv()**

Détermine le lecteur actuel.

```
r% Numéro du lecteur actuel (A=0, B=1, etc...).
```

~GEMDOS(26,L:adr%) **fsetdta()**

Fixe la zone de transfert disque (Disk Transfer Area = DTA), normalement BASEPAGE+128.

```
adr%  Adresse à fixer.
```

r%=GEMDOS(42) **tgetdate()**

Détermine la date système (cf. DATE$).

```
r%   Date : Bits 0 à 4 : jour, 5 à 8 : mois, 9 à 15 : année moins 1980.
```

~GEMDOS(43,d%) **tsetdate()**

Fixe la date système (cf. SETTIME).

```
d%   La nouvelle date (Voyez GEMDOS (42) pour le format).
```

r%=GEMDOS(44) **tgettime()**

Détermine l'heure système (cf. TIME$).

```
r%   L'heure : Bits 0 à 4 : secondes, 5 à 10 : minutes, 11 à 15 : heures.
```

~GEMDOS(45,t%) **tsettime()**

Fixe l'heure système (cf. SETTIME).

```
t%   La nouvelle heure (voyez GEMDOS(44) pour le format).
```

r%=GEMDOS(47) **fgetdta()**

Détermine l'adresse de la zone de transfert disque actuelle (DTA).

```
r%   L'adresse déterminée.
```

r%=GEMDOS(48) **sversion()**

Détermine le numéro de version GEMDOS.

```
r% Le numéro de version.
```

~GEMDOS(49,L:b%,r%) **ptermres()**

Termine le programme et le laisse résident (ne peut être utilisé sous GFA-BASIC).

```
b%  Nombre d'octets à laisser résidents à partir de BASEPAGE.
r%  Valeur renvoyée au programme d'appel.
```

r%=GEMDOS(54,L:adr%,d%) **dfree()**

Obtient des informations sur la place mémoire d'une disquette (cf. DFREE).

```
r%      -46 si un numéro de lecteur incorrect a été spécifié.
adr%    Adresse d'une structure Disk Info de 4 fois 4 octets :
        Long 1 : nombre de clusters libres,
        Long 2 : nombre total de clusters,
        Long 3 : octets par secteur,
        Long 4 : secteurs par cluster.
```

r%=GEMDOS(57,L:adr%) **dcreate()**

Met en place un dossier dans le répertoire actuel (cf. MKDIR).

```
r%      -34 ou -36 si une erreur s'est produite.
adr%    Adresse du nom du nouveau dossier (doit se terminer par un octet nul).
```

r%=GEMDOS(58,L:adr%) **ddelete()**

Supprime un dossier (cf. RMDIR).

```
r%      -34, -36 ou -65 si une erreur s'est produite.
adr%    Adresse du nom du dossier (doit se terminer par un octet nul).
```

r%=GEMDOS(59,L:adr%) **dsetpath()**

Change le répertoire actuel.

```
r%      -34 si nouveau répertoire non trouvé.
adr%    Adresse du nom du répertoire (doit se terminer par un octet nul)
```

r%=GEMDOS(60,L:adr%,a%) **fcreate()**

Crée un nouveau fichier (cf. OPEN "O").

```
r%      -34, -35 ou -36 si une erreur s'est produite.
adr%    Adresse du nom du fichier (doit se terminer par un octet nul).
a%      Bit 0 mis : fichier protégé contre l'écriture,
        Bit 1 : fichier caché,
        Bit 2 : fichier système (y compris caché),
        Bit 3 : nom de disquette.
```

r%=GEMDOS(61,L:adr%,m%) **fopen()**

Ouvre un fichier (cf. OPEN).

```
r%      -34, -35 ou -36 si une erreur s'est produite, sinon handle du fichier
adr%    Adresse du nom du fichier (doit se terminer par un octet nul).
m%      0 pour lecture, 1 pour écriture, 2 pour lecture et écriture.
```

r%=GEMDOS(62,h%) **fclose()**

Fermer un fichier (cf. CLOSE).

```
r%      -37 si une erreur s'est produite.
h%      handle du fichier à fermer.
```

r%=GEMDOS(63,h%,L:1%,L:adr%) **fread()**

Lit des octets sur un fichier ouvert (cf. BGET).

```
r%      -37 si une erreur s'est produite, sinon nombre d'octets lus.
h%      handle du fichier.
l%      nombre d'octets à lire.
adr%    adresse où doivent être écrits les octets lus.
```

r%=GEMDOS(64,h%,L:l%,L:adr%) **fwrite()**

Ecrit des octets dans un fichier (cf. BPUT).

```
r%      -36 ou -37 si une erreur s'est produite, sinon nombre d'octets écrits.
h%      handle du fichier.
l%      nombre d'octets à écrire.
adr%    adresse où figurent dans la mémoire les octets à écrire.
```

r%=GEMDOS(65,L:adr%) **fdelete()**

Supprime un fichier (cf. KILL).

```
r%      -33 ou -36 si une erreur s'est produite.
adr%    Adresse du nom du fichier (doit se terminer par un octet nul).
```

r%=GEMDOS(66,L:n%,h%,m%) **fseek()**

Fixe à nouveau le pointeur pour les accès de fichier (cf. SEEK, RELSEEK).

```
r%      -32 ou -37 si une erreur s'est produite.
n%      Nombre d'octets à sauter.
m%      0 : à partir du début du fichier, 1 : à partir de la position actuelle,
        2 : à partir de la fin du fichier.
```

r%=GEMDOS(67,L:adr%,m%,a%) **fattrib()**

Lit ou écrit des attributs de fichier.

```
r%      -33 ou -34 en cas d'erreur, sinon attributs de fichier jusqu'ici.
adr%    Adresse du nom de fichier (doit se terminer par un octet nul).
a%      Attributs de fichier : Bit 0 : protégé contre l'écriture, 1 : caché,
        2 : fichier système, 3 : nom de disquette, 4 : dossier, 5 : bit d'archivage.
```

r%=GEMDOS(69,h%) **fdup()**

Produit un second handle de fichier.

```
r%      -35 ou -37 si une erreur s'est produite, sinon second handle de fichier.
h%      Le handle de fichier d'origine.
```

r%=GEMDOS(70,h%,nh%) **fforce()**

Fixe un nouveau handle de sortie pour les sorties GEMDOS.

```
r%      -37 si une erreur s'est produite.
h%      Handle du canal de données à rediriger.
nh%     Handle du canal vers lequel les sorties sont redirigées.
```

r%=GEMDOS(71,L:adr%,d%) **dgetpath()**

Détermine le chemin d'accès actuel d'un lecteur (cf. DIR$).

```
r%      -46 si une erreur s'est produite.
adr%    Adresse à partir de laquelle figure le chemin d'accès.
d%      Code du lecteur (0 = actuel, 1 = A, 2 = B, etc...).
```

r%=GEMDOS(72,L:b%) **malloc()**

Réserver ou déterminer place mémoire (cf. MALLOC).

```
r%      Si b%=-1, longueur de la plus grande zone de mémoire libre, sinon adresse
        de départ de la zone réservée ou message d'erreur.
b%      Nombre d'octets à réserver, si b%=-1, voyez r%.
```

r%=GEMDOS(73,L:adr%) **mfree()**

Libère une zone réservée avec GEMDOS(72) (cf. MFREE).

```
r%      -40 si une erreur est apparue.
adr%    Adresse de la zone de mémoire à libérer.
```

r%=GEMDOS(74,L:adr%,L:b%) **mshrink()**

Raccourcit une zone réservée avec GEMDOS(72) (cf. MSHRINK).

```
r%      -40 ou -67 si une erreur est apparue.
adr%    Adresse à partir de la zone de mémoire à raccourcir.
b%      Nouvelle longueur de la zone de mémoire en octets.
```

r%=GEMDOS(75,m%,L:p%,L:c%,L:e%) **pexec()**

Exécute un programme en tant que sous-programme (cf. EXEC).

```
r%      -32, -33, -39 ou -66 en cas d'erreur.
m%      0 : charger et lancer, 3 : charger, 4 : lancer, 5 : produire Base page
p%      Adresse du nom du programme ou de la Base Page si m%=4.
c%      Adresse de la ligne d'instruction (sauf si m%=4)
e%      Adresse de la chaine d'environnement (sauf si m%=4)
```

~GEMDOS(76,r%) **pterm()**

Termine le programme en cours (cf. QUIT, SYSTEM).

```
r%      Valeur de réponse renvoyée au programme d'appel.
```

r%=GEMDOS(78,L:adr%,a%) **fsfirst()**

Recherche dans le répertoire actuel des fichiers dotés de noms déterminés. Le nom de fichier trouvé est placé dans la DTA.

```
r%      -33 : fichier non trouvé, -49 : plus d'autre fichier.
adr%    Adresse du nom de fichier (peut contenir les jokers * et ?).
a%      Attributs de fichier : bit 0 : protégé contre l'écriture, 1 : caché,
        2 : fichier système, 3 : nom de disquette, 4 : dossier, 5 : bit d'archivage.
```

r%=GEMDOS(79) **fsnext()**

Poursuit la recherche entreprise avec GEMDOS(78).

```
r%      -49 si plus d'autre fichier correspondant à la chaine de recherche.
```

r%=GEMDOS(86,0,L:o%,L:n%) **frename()**

Renomme un fichier (cf. NAME, RENAME).

```
r%      -34 ou -36 si une erreur est apparue.
o%      Adresse de l'ancien nom de fichier.
n%      Adresse du nouveau nom de fichier.
```

~GEMDOS(87,L:adr%,h%,m%) **fdatime()**

Fixe ou détermine l'heure et la date d'un fichier (cf. TOUCH).

```
adr%    Adresse des informations d'heure (4 octets).
h%      Handle du fichier.
m%      0 : Lire l'heure du fichier, 1 : Fixer l'heure du fichier.
```

Tables BIOS

~BIOS(0,L:adr%) **getmpb()**

Initialisation du bloc de paramètres de la mémoire (Memory Parameter Block).

```
adr%    Adresse du nouveau MPB.
```

r%=BIOS(1,d%) **constat()**

Déterminer l'état d'entrée d'une unité (cf. INP?).

```
r%  0 : Aucun caractère disponible, -1 caractère disponible.
d%  0 : Interface parallèle, 1 : interface sérielle,
    2 : clavier, 3 : interface MIDI.
```

r%=BIOS(2,d%) **bconin()**

Lit un caractère sur une unité (cf. INP).

```
r%  Caractère lu (8 bits).
d%  0 : Interface parallèle, 1 : interface sérielle,
    2 : clavier, 3 : interface MIDI.
```

~BIOS(3,d%,b%) **bconout()**

Sort un caractère sur une unité (cf. OUT).

```
d%  0 : Interface parallèle, 1 : interface sérielle,
    2 : clavier, 3 : interface MIDI, 4 : IKBD.
b%  Le caractère à sortir.
```

r%=BIOS(4,f%,L:b%,n%,s%,d%) **rwabs()**

Lecture et écriture de secteurs sur la disquette.

```
r%  0 si aucune erreur n'est apparue.
f%  0 : lecture, 1 : écriture.
b%  Adresse de la zone de mémoire des données.
n%  Nombre de secteurs.
s%  Numéro du secteur de départ.
d%  Numéro du lecteur (0 = A, 1 = B, etc...).
```

r%=BIOS(5,n%,L:adr%) **setexec()**

Fixer et lire les vecteurs d'exception.

```
r%      Si adr%=-1, la valeur du vecteur jusqu'ici.
n%      Numéro du vecteur d'exception.
adr%    Nouvelle adresse du vecteur ou -1 (voyez r%).
```

r%=BIOS(6) **tickcal()**

Test du timer système.

```
r%  Nombre de millisecondes écoulées (avec une précision de 20 ms).
```

BIOS(7,d%) **getbpb()**

Détermine l'adresse du bloc de paramètres du BIOS d'un lecteur.

```
r%  Adresse du BPB.
d%  Numéro du lecteur (0=A, 1=B, etc...).
```

r!=BIOS(8,d%) **bcostat()**

Détermine l'état de sortie d'une unité (cf. OUT?).

```
r!  TRUE si caractère peut être envoyé, sinon FALSE.
d%  0 : interface parallèle, 1 : interface sérielle,
    3 : interface MIDI, 4 : chip clavier (IKBD).
```

r%=BIOS(9,d%) **mediach()**

Examine si la disquette a été changée.

```
r%  0 : certainement pas, 1 : peut-être, 2 : changée.
d%  Numéro du lecteur (0 = A, 1 = B, etc...).
```

r%=BIOS(10) **drvmap()**

Examine quels lecteurs sont connectés.

```
r%  Vecteur de bit, un bit mis par lecteur (bit 0 = A, bit 1 = B, etc...).
```

r%=BIOS(11,c%) **kbshift()**

Détermine ou fixe l'état des touches de commutation du clavier.

```
r%  Si c%=-1, état actuel des touches de commutation.
c%  Nouvel état, bit 0 : Shift gauche, bit 1 : Shift droite,
    Bit 2 : Control, bit 3 : Alternate, bit 4 : Caps-Lock,
    bit 5 : Alternate+Clr/Home (bouton droit de la souris),
    bit 6 : Alternate+Insert (bouton gauche de la souris).
```

Tables XBIOS

~XBIOS(0,t%,L:p%,L:v%) **initmous()**

Initialise les routines de traitement de la souris (incompatible avec GEM).

```
t%  Désactiver souris, 1 : activer mode relatif de la souris,
    2 : activer mode absolu de la souris, 4 : souris en mode clavier.
p%  Adresse d'une structure d'informations.
v%  Adresse de la routine de traitement de la souris.
```

r%=XBIOS(2) **physbase()**

Détermine l'adresse de la mémoire écran physique.

```
r%  Adresse de la mémoire écran physique.
```

r%=XBIOS(3) **logbase()**

Déterminer l'adresse de la mémoire écran logique.

```
r%    Adresse de la mémoire écran logique.
```

r%=XBIOS(4) **getrez()**

Détermine la résolution actuelle de l'écran.

```
r%  0 : 320x200, 1 : 640*200, 2 : 640x400, 3 : réservé.
```

~XBIOS(5,L:l%,L:p%,r%) **setscreen()**

Fixe les adresses de la mémoire écran et la résolution, la valeur -1 indiquant pour tous les paramètres l'absence de changement. Les adresses doivent se situer sur des bornes de 256 octets.

```
l%    Nouvelle adresse de la mémoire écran logique.
p%    Nouvelle adresse de la mémoire écran physique.
r%    Nouvelle résolution de l'écran (voyez XBIOS(4), incompatible avec GEM).
```

~XBIOS(6,L:adr%) **setpalette()**

Charge une nouvelle palette de couleurs.

```
adr%  Adresse d'une table de 16 mots contenant la nouvelle palette.
```

r%=XBIOS(7,n%,c%) **setcolor()**

Fixer ou tester un registre de couleur (cf. SETCOLOR).

```
r%    Si c%=-1 test du registre de couleur spécifié.
n%    Numéro du registre de couleur (0 à 15).
c%    Nouvelle couleur, si c%=-1 comme pour r%.
```

r%=XBIOS(8,L:b%,L:f%,d%,sec%,t%,side%,n%) **floprd()**

Lit des secteurs sur la disquette.

```
r%      0 si aucune erreur n'est apparue.
b%      Adresse de la zone dans laquelle les secteurs sont chargés.
```

```
f%      Inutilisé.
d%      Numéro du lecteur (0=A, 1=B, etc...).
sec%    Numéro du secteur de début de lecture.
t%      Numéro de la piste (Track) à lire.
side%   Face de la disquette (0 ou 1).
n%      Nombre de secteurs à lire (doivent figurer sur une même piste).
```

r%=XBIOS(9,L:b%,L:f%,d%,sec%,t%,side%,n%) **flopwr()**

Ecrit des secteurs sur la disquette.

```
r%      0 si aucune erreur n'est apparue.
b%      Adresse de la zone de mémoire contenant les données à écrire.
f%      Inutilisé.
d%      Numéro du lecteur (0=A, 1=B, etc...).
sec%    Numéro du secteur de début l'écriture.
t%      Numéro de la piste (Track) à écrire.
side%   Face de la disquette (0 ou 1).
n%      Nombre de secteurs à écrire (doivent figurer sur une même piste).
```

r%=XBIOS(10,L:b%,L:f%,d%,sec%,t%,side%,i%,L:m%,v%) **flopfmt()**

Formate une piste de la disquette.

```
r%      0 si aucune erreur n'est apparue.
b%      Adresse d'une zone pour les stockages intermédiaires (8 Ko minimum).
f%      Inutilisé.
d%      Numéro du lecteur (0=A, 1=B, etc...).
sec%    Secteurs par piste (normalement 9).
t%      Numéro de la piste (Track) à formater.
side%   Face de la disquette (0 ou 1).
i%      Facteur Interleave (normalement 1).
m%      Numéro magique &H87654321.
v%      Valeur figurant dans les secteurs après formatage (normalement &HE5E5).
```

~XBIOS(12,n%,L:adr%) **midiws()**

Sort une zone de mémoire sur l'interface MIDI.

```
n%      Nombre d'octets à sortir moins 1.
adr%    Adresse de la zone de mémoire à sortir.
```

~XBIOS(13,n%,L:adr%) **mfpint()**

Modifier les vecteurs d'interruption du MFP (ne peut être utilisé en GFA-BASIC).

```
n%      Numéro de l'interruption (0 à 15).
adr%    Nouvelle adresse de l'interruption à régler.
```

r%=XBIOS(14,d%) **iorec()**

Détermine l'adresse des unités d'entrée/sortie.

```
r%      Adresse du buffer de données pour les entrées et sorties de l'unité.
d%      0 : RS 232, 1 : IKBD, 2 : MIDI.
```

~XBIOS(15,b%,h%,ucr%,rsr%,tsr%,scr%) **rsconf()**

Fixe les paramètres pour l'interface sérielle.

```
b%      Vitesse en bauds.
h%      Mode Handshake, 0 : sans, 1 : XON/XOFF, 2 : RTS/CTS, 3 : les deux.
ucr%    Registre de contrôle USART du MFP.
rsr%    Registre d'état du récepteur du MFP.
tsr%    Registre d'état de l'émetteur du MFP.
scr%    Registre de caractère synchrone du MFP.
```

r%=XBIOS(16,L:us%,L:sh%,L:cl%) **keytbl()**

Modification des adresses des tables de définition du clavier.

```
r%      Adresse de la structure KEYTAB.
us%     Adresse de la table pour les touches sans Shift.
sh%     Adresse de la table pour les touches avec Shift.
cl%     Adresse de la table avec Caps-Lock.
```

r%=XBIOS(17) **random()**

Détermine un nombre aléatoire (cf. RAND, RANDOM).

```
r%      Un nombre aléatoire de 24 bits (0 à 16777215).
```

~XBIOS(18,L:b%,L:s%,d%,f%) **protobt()**

Produit un secteur boot sur la disquette.

```
b%      Adresse d'un buffer (512 octets) pour créer le secteur boot.
s%      Numéro de série, -1 : prendre ancien, fixer bit supérieur à 24 : aléatoire.
d%      Type de disquette (pistes/faces), 0 : 40/1, 1 : 40/2, 2 : 80/2, 3 : 80/2).
f%      0 : secteur boot non exécutable, 1 : exécutable, -1 : ne pas modifier.
```

r%=XBIOS(19,L:b%,L:f%,d%,sec%,t%,side%,n%) **flopver()**

Compare les secteurs de la disquette avec le contenu d'une zone de mémoire.

```
b%      Adresse de la zone de mémoire de comparaison.
f%      Inutilisé.
d%      Numéro du lecteur de disquette (0=A, 1=B, etc...).
sec%    Numéro du secteur à partir duquel commence la comparaison.
t%      Numéro de la piste (Track).
side%   Face de la disquette (0 ou 1).
n%      Nombre de secteurs à comparer.
```

~XBIOS(20) **scrdmp()**

Appelle la routine de copie d'écran (cf. HARDCOPY).

r%=XBIOS(21,c%,s%) **curscon()**

Fixe les attributs du curseur.

```
r%      Si c%=5, cela donne la fréquence de clignotement de l'écran.
c%      0 : désactiver le curseur, 1 : activer le curseur, 2 : activer le
        clignotement du curseur, 3 : désactiver le clignotement, 4 : fixer
        fréquence de clignotement sur la valeur dans s%, 5 : voir r%.
s%      Si c%=4, fixer la fréquence de clignotement sur s%.
```

~XBIOS(22,L:t%) **bsettime()**

Fixe la date et l'heure (cf. SETTIME).

```
t%      Bits 0 à 4 : secondes, 5 à 10 : minutes, 11 à 15 : heures, 16 à 20 : jour,
        21 à 24 : mois, 25 à 31 : année moins 1980.
```

r%=XBIOS(23) **bgettime()**

Détermine la date et l'heure (cf. TIME$, DATE$).

```
r%      Définition des bits comme pour XBIOS(22).
```

~XBIOS(24) **bioskey()**

Fixe la définition originelle du clavier (cf. XBIOS(16)).

XBIOS(25,n%,L:adr%) **ikbdws()**

Envoie des données au chip clavier (IKBD).

```
n%      Nombre d'octets à envoyer moins 1.
adr%    Adresse à laquelle figurent les données à envoyer.
```

~XBIOS(26,i%) **jdisint()**

Verrouille une interruption MFP.

```
i%      Numéro de l'interruption à verrouiller (0-15).
```

~XBIOS(27,i%) **jenabint()**

Libère une interruption du MFP.

```
i%      Numéro de l'interruption.
```

r%=XBIOS(28,d%,reg%) **giaccess()**

Lit et écrit des registres du chip son.

```
r%      Si accès en lecture, ici figure la valeur du registre.
d%      Si accès en écriture, ici figure la valeur à écrire (8 bits).
r%      Numéro du registre (0 à 15), accès en écriture si bit 7 mis.
```

~XBIOS(29,b%) **offgibit()**

Annule un bit du registre de PORT A du chip son.

```
b%      Motif de bits combiné par OR avec le motif existant.
```

~XBIOS(30,b%) **ongibit()**

Fixe un bit du registre de PORT A du chip son.

```
b%      Motif de bits combiné par AND avec le motif existant.
```

XBIOS(31,t%,c%,d%,L:adr%) **xbtimer()**

Fixer les timers MFP.

```
t%      Numéro du timer (0 à 3).
c%      Registre de contrôle.
d%      Registre de données.
adr%    Adresse de la routine d'interruption du Timer.
```

~XBIOS(32,L:adr%) **dosound()**

Lance une séquence de son traitée dans l'interruption.

```
adr%    Adresse de la zone de mémoire des instructions Sound.
```

r%=XBIOS(33,c%) **setprt()**

Fixe ou lit les paramètres imprimante.

```
r%      Configuration actuelle, si c%=-1.
c%      Bit non mis/mis, 0 : imprimante matricielle/à marguerite,
        1 : couleur/monochrome, 2 : 1280/960 points par ligne,
        3 : Draft/NLQ, 4 : parallèle/sérielle, 5 : papier continu/feuille à feuille.
```

r%=XBIOS(34) **kbdvbas()**

Adresse d'une table des vecteurs du processeur clavier.

```
r%      Adresse déterminée.
```

r%=XBIOS(35,a%,w%) **kbrate()**

Fixe ou lit la fréquence de répétition du clavier.

```
r%      Données actuelles, bits 0 à 7 : fréquence de répétition,
        8 à 15 : durée de prise en compte.
a%      Délai de prise en compte.
w%      Fréquence de répétition.
```

~XBIOS(36,L:adr%) **prtblk()**

Routine de copie d'écran avec bloc de paramètres.

```
adr%    Adresse d'un bloc de paramètres pour la routine de copie d'écran.
```

~XBIOS(37) **vsync()**

Attend la prochaine interruption Vertical Blank (cf. VSYNC).

~XBIOS(38,L:adr%) **supexec()**

Appel d'une routine assembleur en mode Supervisor (sans appel système).

```
adr%    Adresse de la routine assembleur.
```

~XBIOS(39) **puntaes()**

Désactive l'AES si elle ne figure pas en ROM.

r%=XBIOS(64,b%) **blitmode()**

Commande et test du blitter (seulement sous le TOS du blitter).

```
r%      Etat blitter actuel, si b%=-1, bit 1 : blitter présent ?.
b%      -1 : voyez r%, sinon bit 0 : mis blitter activé, sinon blitter désactivé,
        bits 1 à 14 : réservés (-1), bit 15 : 0.
```

Table des variables Line-A

L-A renvoie l'adresse de base des variables Line-A :

(L-A-906)	Adresse du header de fonte actuel.
L-A-856 ..	37 mots, DEFMOUSE activé.
(L-A-460)	Pointeur sur le header de fonte actuel.
L-A-456	Tableau de quatre pointeurs dont le dernier doit être nul. Chaque pointeur désigne une liste de jeux de caractères formant une chaine. Les deux premiers pointeurs désignent les fontes résidentes. Le troisième désigne les fontes GDOS, il est annulé lors de chaque appel VDI.
INT{L-A-440}	Nombre total de ces fontes.
INT{L-A-46}	Hauteur de ligne de texte.
INT{L-A-44}	Colonne limite du curseur.
INT{L-A-42}	Ligne limite du curseur.
INT{L-A-40}	Longueur d'une ligne de texte en octets.
INT{L-A-38}	Couleur de fond du texte.
INT{L-A-36}	Couleur de premier plan du texte.
{L-A-34}	Adresse du curseur dans la mémoire écran.
INT{L-A-30}	Distance de la première ligne de texte au bord supérieur de l'écran.
INT{L-A-28}	CRSCOL.
INT{L-A-26}	CRSLIN.
BYTE{L-A-24}	Période de clignotement du curseur.
BYTE{L-A-23}	Compteur de clignotements du curseur.
{L-A-22}	Adresse des données de fonte pour le mode monochrome.
INT{L-A-18}	Dernier caractère ASCII de la fonte.
INT{L-A-16}	Premier caractère ASCII de la fonte.
INT{L-A-12}	Résolution horizontale en points écran.
{L-A-10}	Adresse de la table d'offsets de fonte.
INT{L-A-4}	Résolution verticale en points écran.
INT{L-A-0}	Nombre de plans de bits.
INT{L-A+2}	Octets par ligne écran.
{L-A+4}	Pointeur sur le tableau CONTRL.
{L-A+8}	Pointeur sur le tableau INTIN.
{L-A+12}	Pointeur sur le tableau PTSIN.
{L-A+16}	Pointeur sur le tableau INTOUT.
{L-A+20}	Pointeur sur le tableau PTSOUT.
INT{L-A+24}	Valeur de couleur pour le niveau de bits 0.
INT{L-A+26}	Valeur de couleur pour le niveau de bits 1.
INT{L-A+28}	Valeur de couleur pour le niveau de bits 2.
INT{L-A+30}	Valeur de couleur pour le niveau de bits 3.
INT{L-A+32}	Flag, ne pas dessiner dernier point écran d'une ligne
INT{L-A+34}	Motif de ligne.

INT(L-A+36)	Mode graphique.
INT(L-A+38) à	
INT(L-A+44)	2 couples de coordonnées.
(L-A+46)	Pointeur sur le motif de remplissage actuel.
(L-A+50)	Pointeur sur le masque de motif de remplissage actuel.
INT(L-A+52)	Flag pour remplissage polychrome.
INT(L-A+54)	Flag de Clipping.
INT(L-A+56) à	
INT(L-A+64)	Coordonnées de Clipping.
INT(L-A+66)	Facteur d'agrandissement.
INT(L-A+68)	Direction d'agrandissement.
INT(L-A+70)	Flag pour impression proportionnelle.
INT(L-A+72)	Offset x pour Textblt.
INT(L-A+74)	Offset y pour Textblt.
INT(L-A+76)	Coordonnée x d'un caractère sur l'écran.
INT(L-A+78)	Coordonnée y d'un caractère sur l'écran.
INT(L-A+80)	Largeur d'un caractère.
INT(L-A+82)	Hauteur d'un caractère.
(L-A+84)	Pointeur sur Image de jeu de caractères.
(L-A+88)	Largeur de l'image de jeu de caractères.
INT(L-A+90)	Style du texte.
INT(L-A+92)	Masque pour la sortie de texte avec ombre.
INT(L-A+94)	Masque pour l'écriture en italique.
INT(L-A+96)	Largeur supplémentaire pour les caractères gras.
INT(L-A+98)	Offset droit italique.
INT(L-A+100)	Offset gauche italique.
INT(L-A+102)	Flag d'agrandissement.
INT(L-A+104)	Angle de rotation du texte.
INT(L-A+106)	Couleur du texte.
(L-A+108)	Pointeur sur buffer pour les effet de texte.
INT(L-A+112)	Offset pour un second buffer de ce type.
INT(L-A+114)	Couleur de fond du texte.
INT(L-A+116)	Flag pour copy raster form, <>0 pour transparent.
(L-A+118)	Pointeur sur une routine permettant l'arrêt d'une opération de remplissage ; sous 3.0 modifié par Shift-Alternate-Control.

Table des paramètres d'entrée pour V_OPN(v)WK

Optionnel sauf x, avec les valeurs défaut suivantes :

x Numéro d'identification de périphérique (Standard)

```
 1 .. : écran            11 .. : plotter
21 .. : imprimante       31 .. : fichier Meta
41 .. : caméra           51 .. : tableur graphique
```

```
1  Type de ligne.
1  Couleur de ligne.
1  Type de marquage.
1  Couleur de marquage.
1  Style de texte.
1  Couleur de texte.
1  Type de remplissage.
1  Style de remplissage.
1  Couleur de remplissage.
2  Système de coordonnées (0 : NDC, 1 : réservé, 2 : RC)
```

Table du tableau WORK_OUT de VDI

Pour V_OPN(v)WK, les valeurs de résultat sont dans INTOUT(0) à INTOUT(44) et dans PTSOUT(0) à PTSOUT(11).

```
WORK_OUT(0)      Largeur maximale en points écran.
WORK_OUT(1)      Hauteur maximale en points écran.
WORK_OUT(2)      0 : graduation précise de l'image possible,
                 1 : impossible.
WORK_OUT(3)      Largeur d'un point écran en micromètres.
WORK_OUT(4)      Hauteur d'un point écran en micromètres.
WORK_OUT(5)      Nombre de hauteurs de caractère (0 : modifiable).
WORK_OUT(6)      Nombre de types de ligne.
WORK_OUT(7)      Nombre de largeurs de ligne (0 : modifiable).
WORK_OUT(8)      Nombre de symboles de marquage.
WORK_OUT(9)      Nombre de tailles de symboles de marquage (0 : modifiable).
WORK_OUT(10)     Nombre de jeux de caractères.
WORK_OUT(11)     Nombre de motifs.
WORK_OUT(12)     Nombre de motifs de hachures.
WORK_OUT(13)     Nombre de couleurs prédéfinies.
WORK_OUT(14)     Nombre de fonctions graphiques de base (GDP).
```

WORK_OUT(15) à
WORK_OUT(24) Liste des fonctions graphiques de base (GDP). Dix fonctions de base sont soutenues :
1 : bar 2 : arc 3 : pie
4 : circle 5 : ellipse 6 : elliptical arc
7 : elliptical pie 8 : rounded rectangle
9 : filled rounded rectangle
10 : justified graphic text
La fin de cette liste est marquée par -1.

WORK_OUT(25) à
WORK_OUT(34) Liste des attributs des fonctions graphiques de base :
0 : ligne 1 : marqueur 2 : texte
3 : zone pleine 4 : pas d'attribut.

WORK_OUT(35) 0 : les couleurs ne peuvent être représentées,
1 : peuvent être représentées.

WORK_OUT(36) 0 : texte ne peut subir de rotation,
1 : peut subir une rotation.

WORK_OUT(37) 0 : les fonctions de remplissage ne sont pas possibles,
1 : possibles.

WORK_OUT(38) 0 : CELLARRAY non disponible, 1 : disponible.

WORK_OUT(39) Nombre de couleurs pouvant être représentées :
0 : plus de 32767, 1 : monochrome,
supérieure à 2 : nombre de couleurs.

WORK_OUT(40) 1 : positionnement du curseur graphique avec clavier uniquement,
2 : avec clavier et souris.

WORK_OUT(41) Périphérique d'entrée de valeurs : 1 : clavier,
2 : autre périphérique.

WORK_OUT(42) Touches de sélection : 1 : touches de fonction,
2 : autres touches.

WORK_OUT(43) Nombre de périphériques d'entrée de chaines :
1 : clavier.

WORK_OUT(44) Type de station de travail : 0 : sortie seulement,
1 : entrée seulement, 2 : entrée/sortie,
3 : réservé, 4 : fichier Meta.

WORK_OUT(45) Largeur minimale de caractère.

WORK_OUT(46) Hauteur minimale de caractère.

WORK_OUT(47) Largeur maximale de caractère.

WORK_OUT(48) Hauteur maximale de caractère.

WORK_OUT(49) Largeur de ligne visible minimale.

WORK_OUT(50) Réservé, toujours 0.

WORK_OUT(51) Largeur de ligne maximale dans le sens des X.

WORK_OUT(52) Réservé, toujours 0.

WORK_OUT(53) Largeur de marquage minimale.

WORK_OUT(54) Hauteur de marquage minimale.

WORK_OUT(55) Largeur de marquage maximale.

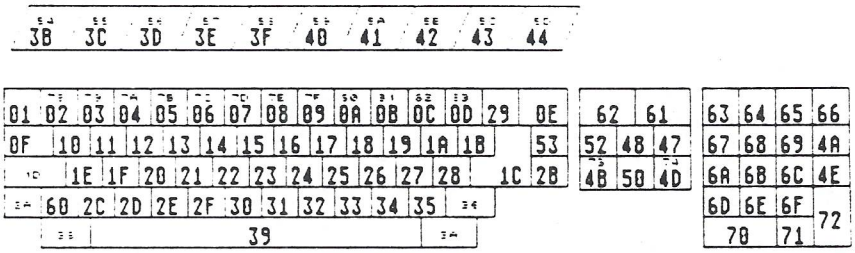
WORK_OUT(56) Hauteur de marquage maximale.

Table VT 52

Les routines de l'émulateur VT 52 peuvent être appelée en sortant avec PRINT les chaînes de la table suivante.

```
CHR$(27)+"A";  Curseur vers le haut (s'arrête sur le bord supérieur).
CHR$(27)+"B";  Curseur vers le bas (s'arrête sur le bord inférieur).
CHR$(27)+"C";  Curseur vers la droite (s'arrête sur le bord droit).
CHR$(27)+"D";  Curseur vers la gauche (s'arrête sur le bord gauche).
CHR$(27)+"E";  Vider écran (CLS).
CHR$(27)+"H";  Curseur dans le coin supérieur gauche de l'écran (LOCATE 1,1).
CHR$(27)+"I";  Curseur vers le haut (scrolling sur le bord supérieur).
CHR$(27)+"J";  Vide l'écran à partir de la position du curseur.
CHR$(27)+"K";  Efface la ligne à partir de la position du curseur.
CHR$(27)+"L";  Insère une ligne vide dans l'emplacement du curseur.
CHR$(27)+"M";  Efface une ligne dans l'emplacement du curseur (le reste est ramené
                 vers le haut).
CHR$(27)+"Y"+CHR$(c+32)+CHR$(l+32);  Equivaut à LOCATE l,c.
CHR$(27)+"b"+CHR$(c);  Sélectionne c comme couleur d'écriture.
CHR$(27)+"c"+CHR$(c);  Sélectionne c comme couleur du fond.
CHR$(27)+"d";  Vide l'écran jusqu'à l'emplacement du curseur.
CHR$(27)+"e";  Activer curseur.
CHR$(27)+"f";  Désactiver curseur.
CHR$(27)+"j";  Sauvegarder position du curseur.
CHR$(27)+"k";  Fixer curseur sur la position sauvegardée avec CHR$(27)+"j".
CHR$(27)+"l";  Effacer ligne dans laquelle figure le curseur.
CHR$(27)+"o";  Effacer ligne jusqu'à l'emplacement du curseur.
CHR$(27)+"p";  Activer écriture en inversion vidéo.
CHR$(27)+"q";  Désactiver écriture en inversion vidéo.
CHR$(27)+"v";  Activer débordement de ligne automatique.
CHR$(27)+"w";  Désactiver débordement de ligne automatique.
```

Table des codes clavier



Caractères ASCII spéciaux

Appellation des codes ASCII de 0 à 31.

0	NUL	1	SOH	2	STX	3	ETX
4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	LF	11	VT
12	FF	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3
20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC
28	FS	29	GS	30	RS	31	US
127	DEL						

Table des codes ASCII

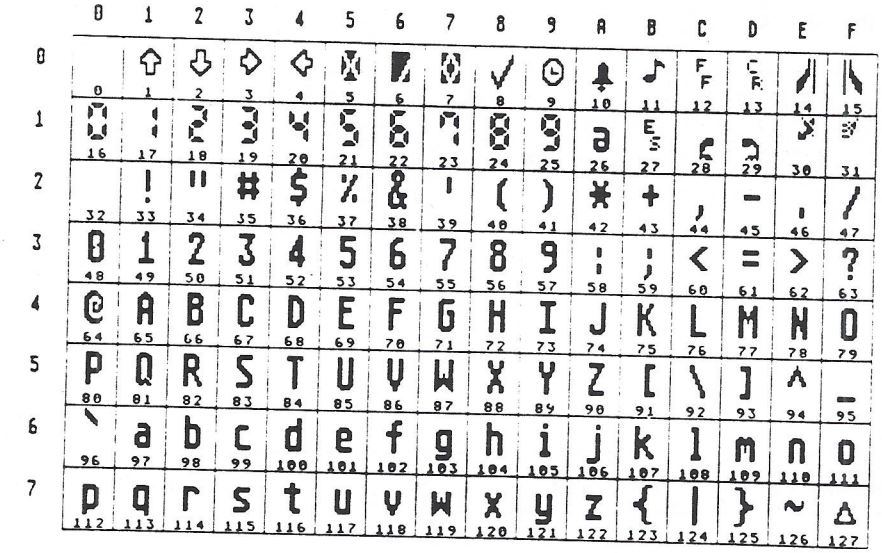
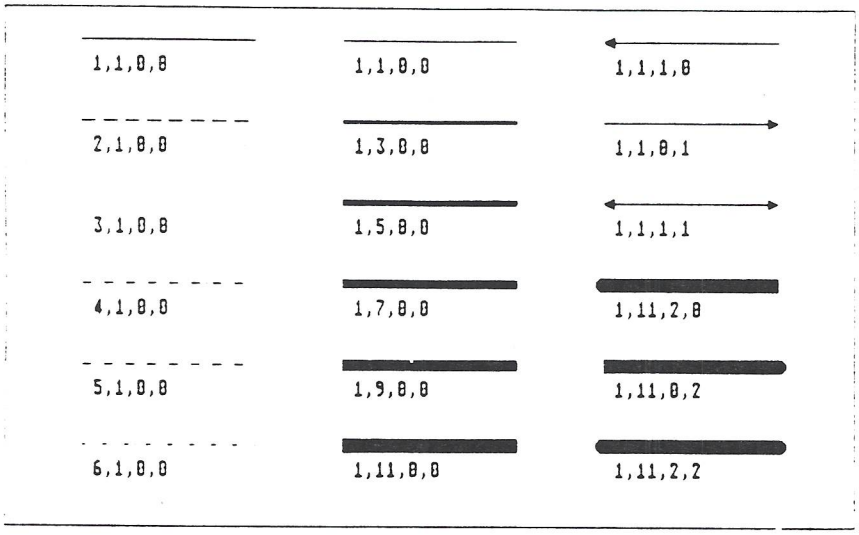
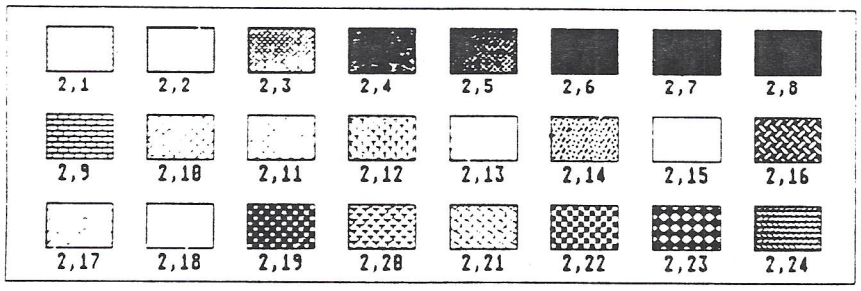
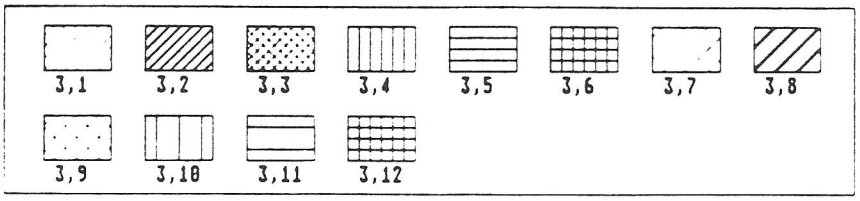


Table des motifs de remplissage et styles de ligne







Messages d'erreur

Messages d'erreur GFA-BASIC

```
 0 Division par zéro
 1 Débordement
 2 Le nombre n'est pas un Integer|-2147483648 .. 2147483647
 3 Le nombre n'est pas un octet|0 .. 255
 4 Le nombre n'est pas un mot|-32768 .. 32767
 5 Racine carrée d'un nombre|négatif impossible
 6 Logarithme d'un nombre|inférieur à zéro impossible
 7 Erreur inconnue
 8 Mémoire pleine
 9 Fonction ou instruction|impossible
10 Chaine trop longue|max 32767 caractères
11 Le programme n'est pas|en GFA BASIC 3.0
12 Programme trop grand|Mémoire pleine|NEW
13 Le fichier programme|n'est pas en GFA BASIC
14 Champ dimensionné deux fois
15 Champ non dimensionné
16 Index de champ trop grand
17 Index de dim trop grand
18 Mauvais nombre d'indices
19 Procédure introuvable
20 Marque introuvable
21 Pour OPEN utiliser|"I"nput  "O"utput "R"andom|"A"ppend
22 Fichier déjà ouvert
23 Mauvais numéro de fichier
24 Fichier non ouvert
25 Mauvaise saisie|Ce n'est pas un nombre
26 Fin de fichier atteinte|EOF
27 Trop de points pour|Polyline/Polyfill|max 128
28 Le champ ne peut avoir qu'une dimension
29 Nombre de points plus grand que le champ
30 Merge, ce n'est pas|un fichier ASCII
31 Merge, ligne trop longue - Arrêt
32 ==> Syntax incorrecte|arrêt du programme
33 Marque non définie
34 Trop peu de données
35 Donnée non numérique
37 Disquette pleine
38 Instruction impossible|en mode direct
39 Erreur de programmation|GOSUB impossible
40 CLEAR n'est pas possible dans|une boucle FOR NEXT|ou une procédure
```

```
41 CONT impossible
42 Trop peu de paramètres
43 Expression trop complexe
44 Fonction indéfinie
45 Trop de paramètres
46 Paramètre inexact|ce doit être un nombre
47 Paramètre inexact|ce doit être une chaine
48 Open "R" - Enregistrement trop long
49 Trop de fichiers "R"
50 Pas de fichier "R"
52 Champ plus grand que l'enregistrement
54 Mauvaise longueur|d'enregistrement GET/PUT
55 Mauvais numéro|de phrase GET/PUT
60 Longueur de chaine|de SPRITE erronée
61 RESERVE Erreur
62 Erreur dans menu
63 Erreur dans Reserve
64 Erreur dans pointeur
65 Champ < 256
66 VAR-champ ?
67 Erreur ASIN/ACOS
68 VAR unpaired
69 ENDFUNC sans RETURN
71 Index trop grand
90 Erreur dans Local
91 Erreur dans For
92 Resume (next) impossible|fatal, For ou Local
93 Stack Erreur
```

Messages d'erreur de bombes

```
100 GFA-BASIC Version 3.00| Copyright 1986-1988|GFA Systemtechnik GmbH
102 2 bombes - erreur bus|peut-être mauvais Peek ou Poke
103 3 bombes - erreur d'adresse|adresse de mot impaire|avec Dpoke/Dpeek,
               Lpoke|ou Lpeek?
104 4 bombes - exécution d'une instruction 68000|ne convenant pas
105 5 bombes - division par zéro|en langage machine 68000
106 6 bombes - exception CHK|interruption 68000|par instruction CHK
107 7 bombes - exception TRAPV|interruption 68000|par instruction TRAPV
108 8 bombes - interruption 68000|par exécution d'une|instruction privilégiée
109 9 bombes - exception trace|interruption trace avec 68000
```

Messages d'erreur du TOS

```
 -1 * Erreur générale
 -2 * Drive not Ready|dépassement de temps imparti
 -3 * Erreur inconnue
 -4 * Erreur CRC|Test de somme de contrôle disque faux
 -5 * Bad Request|Instruction incorrecte
 -6 * Seek Error|Piste non trouvée
 -7 * Unknown Media|Secteur boot incorrect
 -8 * Secteur non trouvé
 -9 * Plus de papier
-10 * Erreur en écriture
-11 * Erreur de lecture
-12 * Erreur générale 12
-13 * Disquette protégée contre l'écriture
-14 * Disquette a été changée
-15 * Périphérique inconnu
-16 * Bad Sector (Verify)
-17 * Introduire une autre disquette
-32 * Numéro de fonction incorrect
-33 * Fichier non trouvé
-34 * Nom de chemin non trouvé
-35 * Trop de fichiers ouverts
-36 * Accès impossible
-37 * Handle incorrect
-39 * Mémoire pleine
-40 * Adresse de bloc de mémoire incorrecte
-46 * Désignation de lecteur incorrecte
-49 * Plus d'autres fichiers
-64 * Erreur de zone GEMDOS|Seek incorrect ?
-65 * Erreur GEMDOS interne
-66 * Ce n'est pas un programme binaire
-67 * Erreur de bloc de mémoire
```

Messages d'erreur de l'éditeur

```
Case sans Select
Select sans endselect
While sans Wend
Repeat sans Until
Do sans Loop
For sans next
Wend sans While
Until sans Repeat
Loop sans Do
Next sans For
```

If sans Endif
Endif sans If
Else sans If
Else sans Endif
Exit sans boucle
Procedure sans Return
Procedure dans boucle
Procedure définie deux fois
Function sans Endfunc
Function dans boucle
Function définie deux fois
Return sans Procedure
Marke définie deux fois
Local seulement dans Procedure
Local non autorisé dans boucle
Function définie deux fois
Goto vers/de For-Next ou Procedure
Resume dans boucle For-Next
Resume sans Procedure
Pas de Resume dans Function
Endfunc sans Function

LISTE ALPHABETIQUE DES FONCTIONS

!....25
#....25
$....24, 25
%....25
&....25
*....35, 57
+....57
-....57
/....57
<....65
=....65
>....65
@....157
\....57
^....57
|....25
~....49

A

ABS....71
ABSOLUTE....40
ACHAR....233
ACLIP....225
ACOS....76
ADD....80, 81
ADDRIN....251
ADDROUT....251
AFTER....162
AFTER CONT....162
AFTER STOP....162
ALERT....219
ALINE....226
AND....60, 86
APOLY....228
APPL....258
APPL_EXIT....260
APPL_FIND....259
APPL_TPLAY....260
APPL_TRECORD....260
APPL_WRITE....259

ARECT....228
ARRAYFILL....29
ARRPTR....39
ASC....30
ASIN....76
ATEXT....233
ATN....76

B

BASEPAGE....51
BCHG....83
BCLR....83
BGET....122
BIN$....31
BIOS....223
BITBLT....229
BLOAD....122
BMOVE....50
BOUNDARY....180
BOX....189
BPUT....122
BSAVE....122
BSET....83
BTST....83
BYTE....37, 87

C

C:....244
CALL....246
CARD....37, 87
CASE....145
CFLOAT....34
CHAIN....168
CHAR....37
CHDIR....114
CHDRIVE....114
CHR$....30
CINT....34
CIRCLE....190
CLEAR....41
CLEARW....215
CLIP....185
CLIP OFF....185

CLIP OFFSET 185
CLOSE 120
CLOSEW 213
CLR 41
CLS 193
COLOR 176
CONT 145
CONTRL 236
COS 76
COSQ 76
CRSCOL 105
CRSLIN 105
CVD 33
CVF 33
CVI 33
CVL 33
CVS 33

D

DATA 111
DATE 46, 47
DEC 79
DEFAULT 145
DEFBIT 22
DEFBYT 22
DEFFILL 179
DEFFLT 22
DEFFN 158
DEFINT 22
DEFLINE 181
DEFLIST 23
DEFMARK 178
DEFMOUSE 176
DEFNUM 104
DEFSTR 22
DEFTEXT 182
DEFWRD 22
DEG 76
DELAY 165
DELETE 45
DFREE 114
DIM 27
DIM? 27
DIR 115
DIR$ 114

DIV....57, 80, 81
DO....152
DO UNTIL....152
DO WHILE....152
DOUBLE....37
DOWNTO....149
DPEEK....36
DPOKE....36
DRAW....186
DRAW TO....186
DUMP....172

E

EDIT....166
ELSE....142
ELSEIF....143
END....166
ENDFUNC....157
ENDIF....142
ENDSELECT....145
EOF....120
EQV....64, 86
ERASE....41
ERR....161
ERR$....161
ERROR....161
EVEN....72
EVERY....162
EVERY CONT....162
EVERY STOP....162
EVNT_BUTTON....261
EVNT_DCLICK....265
EVNT_KEYBD....260
EVNT_MESAG....263
EVNT_MOUSE....262
EVNT_MULTI....264
EVNT_TIMER....263
EXEC....248
EXIST....118
EXIT IF....154
EXP....75

F

FALSE....46
FATAL....161
FGETDTA....116
FIELD....129
FIELD AS....129
FILES....115
FILESELECT....220
FILL....192
FIX....73
FLOAT....37
FN....158
FOR....149
FORM INPUT....100
FORM_ALERT....272
FORM_BUTTON....273
FORM_CENTER....273
FORM_DIAL....271
FORM_DO....271
FORM_ERROR....272
FORM_KEYBD....273
FRAC....73
FRE....50
FSEL_INPUT....280
FSETDTA....116
FSFIRST....117
FSNEXT....117
FULLW....215
FUNCTION....157

G

GB....251
GCONTRL....251
GDOS....239
GEMDOS....223
GEMSYS....252
GET....130, 195
GINTIN....251
GINTOUT....251
GOSUB....155
GOTO....165
GRAF_DRAGBOX....275
GRAF_GROWBOX....276
GRAF_HANDLE....278

GRAF_MKSTATE....279
GRAF_MOUSE....278
GRAF_MOVEBOX....275
GRAF_RUBBERBOX....274
GRAF_SHRINKBOX....276
GRAF_SLIDEBOX....277
GRAF_WATCHBOX....277
GRAPHMODE....184

H

HARDCOPY....137
HEX$....31
HIDEM....134
HIMEM....51
HLINE....227
HTAB....105

I

IF....142
IMP....63, 86
INC....79
INFOW....215
INKEY....97
INLINE....53
INP....123, 131
INPAUX....132
INPMID....132
INPUT....98, 124
INPUT$....124
INSERT....45
INSTR....92
INT....37, 73
INTIN....236
INTOUT....236

K

KEYDEF....109
KEYGET....107
KEYLOOK....107
KEYPAD....106
KEYPRESS....108

KEYTEST 107
KILL 121

L

L 224
LEFT$ 89
LEN 91
LET 48
LINE 186
LINE INPUT 99, 124
LINE-A 225
LIST 168
LLIST 168
LOAD 167
LOC 120
LOCAL 156
LOCATE 101
LOF 120
LOG 75
LOG10 75
LONG 37
LOOP 152
LOOP UNTIL 152
LOOP WHILE 152
LPEEK 36
LPOKE 36
LPOS 137
LPRINT 137
LSET 94

M

MALLOC 54
MAX 74
MENU 202, 209
MENU KILL 210
MENU OFF 210
MENU_BAR 265
MENU_ICHECK 265
MENU_IENABLE 266
MENU_REGISTER 267
MENU_TEXT 266
MENU_TNORMAL 266
MFREE 54

MID$....90, 94
MIN....74
MKD....33
MKDIR....118
MKF....33
MKI$....33
MKL....33
MKS....33
MOD....57, 81
MODE....104
MONITOR....246
MOUSE....133
MOUSEK....133
MOUSEX....133
MOUSEY....133
MSHRINK....54
MUL....80, 81

N

NAME....121
NEW....167
NEXT....149
NOT....59
NUMLOCK....12

O

OB_ADR....255
OB_FLAGS....255
OB_H....255
OB_HEAD....255
OB_NEXT....255
OB_SPEC....255
OB_STATE....255
OB_TAIL....255
OB_TYPE....255
OB_W....255
OB_X....255
OB_Y....255
OBJC_ADD....267
OBJC_CHANGE....270
OBJC_DELETE....268
OBJC_DRAW....268
OBJC_EDIT....270

OBJC_FIND....268
OBJC_OFFSET....269
OBJC_ORDER....269
OCT$....31
ODD....72
ON BREAK....159
ON BREAK CONT....159
ON BREAK GOSUB....159
ON ERROR....160
ON ERROR GOSUB....160
ON MENU....201
ON MENU BUTTON....205
ON MENU GOSUB....209
ON MENU IBOX....207
ON MENU KEY GOSUB....206
ON MENU MESSAGE GOSUB....208
ON MENU OBOX....207
ON x GOSUB....145
OPEN....119
OPENW....213
OPTION BASE....28
OR....61, 86
OUT....123, 131

P

PAUSE....165
PBOX....189
PCIRCLE....190
PEEK....36
PELLIPSE....190
PI....46
PLOT....186
POKE....36
POLYFILL....191
POLYLINE....191
POLYMARK....191
POS....105
PRBOX....189
PRED....81, 90
PRINT....101, 125
PRINT AT....101
PRINT USING....102, 125
PROCEDURE....154, 155
PSAVE....167
PSET....225

PTSIN....236
PTSOUT....236
PTST....226
PUT....130, 195

Q

QSORT....43
QUIT....169

R

RAD....76
RAND....77
RANDOM....77
RANDOMIZE....77
RBOX....189
RC....219
RC_INTERSECT....218
RCALL....247
READ....111
RECALL....126
RECORD....130
RELSEEK....127
REM....164
RENAME....121
REPEAT....150
RESERVE....52
RESTORE....111
RESUME....160
RESUME [NEXT]....160
RETURN....155, 157
RIGHT$....89
RINSTR....92
RMDIR....118
RND....77
ROL....84
ROR....84
ROUND....74
RSET....94
RSRC_FREE....287
RSRC_GADDR....288
RSRC_LOAD....287
RSRC_OBFIX....289

RSRC_SADDR....288
RUN....169

S

SAVE....167
SCRP_READ....279
SCRP_WRITE....280
SDPOKE....36
SEEK....127
SELECT....145
SETCOLOR....176
SETDRAW....187
SETMOUSE....134
SETTIME....46, 47
SGET....195
SGN....71
SHEL_ENVRN....292
SHEL_FIND....292
SHEL_GET....291
SHEL_PUT....291
SHEL_READ....290
SHEL_WRITE....290
SHL....84
SHOWM....134
SHR....84
SIN....76
SINGLE....37
SINQ....76
SLPOKE....36
SOUND....138
SPACE$....93
SPC....93
SPOKE....36
SPRITE....194
SPUT....195
SQR....75
SSORT....43
STICK....135
STOP....166
STORE....126
STR$....31
STRIG....135
STRING$....93
SUB....80, 81
SUCC....81, 90

SWAP..... 42, 87
SYSTEM..... 169

T

TAB..... 105
TAN..... 76
TEXT..... 193
TIME..... 46, 47
TIMER..... 46, 48
TITLEW..... 215
TOPW..... 215
TOUCH..... 120
TRACE..... 171
TRIM$..... 91
TROFF..... 170
TRON..... 170
TRUE..... 46
TRUNC..... 73
TYPE..... 29

U

UNTIL..... 150
UPPER$..... 93

V

V:..... 39
V_OPNWK..... 240
VAL..... 32
VAL?..... 32
VARPTR..... 39
VDIBASE..... 237
VDISYS..... 236
VOID..... 49
VQT..... 243
VST..... 242
VSYNC..... 196
VTAB..... 105

W

W:..... 224
W_HAND..... 214
W_INDEX..... 214
WAVE..... 138
WEND..... 151
WHILE..... 151
WIND_CALC..... 286
WIND_CLOSE..... 282
WIND_CREATE..... 281
WIND_DELETE..... 282
WIND_FIND..... 285
WIND_GET..... 283
WIND_OPEN..... 282
WIND_SET..... 284
WIND_UPDATE..... 286
WINDTAB..... 216
WITH..... 43
WORD..... 87
WORK_OUT..... 238
WRITE..... 101, 125

X

XBIOS..... 223
XOR..... 62, 86

INDEX

A

Accès séquentiel indexé....127
Adresse de la Basepage....51
Adresses des blocs de paramètres AES....251
Adresses des blocs de paramètres VDI....236
Afficher le contenu de variables....172
Allouer des zones de mémoire....54
Ancienne version....16
Apparence d'une chaîne de caractères....182
Appeler une routine AES....252
Appels LINE-A....225
Application library....258
Arc d'ellipse....190
Arithmétique entière....79
Arrêt du programme....160

B

Bibliothèque AES....251
Bloc de touches numériques....11
Boucle de comptage....149
Boucles....149
Bouton défaut....220
Buffer de message....202

C

Cercle....190
Chaînes de caractères....26
Champ....128
Changer de lecteur....114
Charger des données à partir d'un fichier....124
Charger un programme....167
Chemin d'accès....113
Clé de tri....44
Clipping....185
Code de l'événement....202
Combinaisons logiques....86
Combiner des chaînes de caractères....65

Commentaire 164
Constante PI 47
Coordonnées de joystick 135
Copie d'écran 137
Copier des sections rectangulaires de l'écran 197
Copier des zones de mémoire 50
Couleur du point 191
Curseur de souris personnel 177

D

Date système 47
Début de bloc 18
Décaler des bits 83
Déclaration simplifiée des variables 22
Déclarations globales de variables 23
Définir des fonctions sur une ligne 158
Définir le format des chaînes de caractères 103
Définir le format des expressions numériques 102
Définir les touches de fonctions 109
Déplacements du curseur 10
Désactiver le curseur de la souris 134
Descripteur de tableau 26
Dessin relatif 187
Déterminer le signe d'une expression 72
Dimensions du tableau 27

E

Editeur 9
Eléments d'indice 28
Ellipse 190
Emploi de virgules 101
Emploi des parenthèses 70
Enregistrement 128
Entrée de l'heure 22
Entrées du menu 202
Entrées et sorties générales 111
Entrer des variables 98
Erreur 160
Etat des boutons de la souris 133
Evénements simples 201
Event library 260
Examen de la structure 21
Extension .BAK 16

F

Fenêtres....212
Fichier relatif....128
Fichier séquentiel....128
Fin d'un fichier....120
Fin de bloc....18
Fixer le mode Deflist....15
Folding....10
Fonctions VDI....234
Form library....271
Format d'affichage de la date....104

G

Générateur de nombres aléatoires....78
Générer des sons....139
Gestion d'un menu déroulant....209
Gestion des fenêtres....212
Gestion du curseur....9

H

Handle GEM de la fenêtre....214
Heure système....47
Hiérarchie des opérateurs....69

I

Impression du programme....17
Insérer une expression de chaîne....94
Insert/Overwr....20
Instructions Control....12
Instructions de point....17
Instructions KEYXXX....106
Interfaces sérielle et MIDI....132
Interpréteur RUN-ONLY....302
Interrompre un programme....21

J

Joystick....133

L

Libérer une zone de mémoire..... 52
Ligne..... 186
LINE-A..... 225
Lire le buffer clavier..... 107
Logarithme..... 75
Longueur d'un fichier..... 120
Longueur d'une chaîne..... 89

M

Manette de jeux..... 135
Manipulation des fichiers..... 128
Manipulations des adresses..... 36
Marque..... 165
Marques sous l'éditeur..... 14
Mémoire à libérer..... 54
Mémoire libre..... 50
Menu GFA-BASIC..... 15
Menu library..... 265
Menus déroulants..... 209
Mode direct..... 21
Modes graphiques..... 175
Montre..... 22
Motif de remplissage..... 179

N

Nombres aléatoires..... 78
Nouveaux noms..... 15
Numéro de ligne..... 22
Numérotation des lignes..... 17
NUMLOCK..... 12

O

Opérateur d'affectation..... 68
Opérateurs arithmétiques..... 57
Opérateurs logiques..... 58
Organisation des fichiers..... 113

P

Pg Down....20
Pg Up....20
Point....186
Point d'action....177
Pointeur....35
Pointeur de données....112
Points du menu....16
Polygone avec un motif....191
Position de tabulation....12
Possibilités de sortie....97
Procédure....154
Procédures mémorisées....10
Programmation des interruptions....162
Programmes des anciennes versions....16

Q

Quick sort....43
Quitter le mode direct....21

R

Racine....113
Racine carrée....75
Recherche....19
Rectangle....189
Rectangle plein....189
Registre de couleur....176
Remplacer un texte....19
Remplissage....192
Répertoires....113
Resource library....287
Rotation de bits....83
Routines système....223

S

Saut de ligne....101
Sélecteur d'alerte....219
Sélecteur d'objet....220
Shell sort....43
Sous-programmes....154

Spécifier des valeurs constantes....111
Stocker des données....101
Structure d'objet....253
Style de ligne....181
Supprimer des tableaux....41
Supprimer toutes les variables....41
Supprimer un élément d'un tableau....45
Surveillance du clavier....206
Symbole Atari....14
Symbole de pointeur....35
Synchroniser la construction de l'écran....196

T

Table de fenêtre....216
Table de paramètres....216
Tableaux....26
Temps écoulé....48
Terminer un programme....166
Touche Delete....12
Touche Help....10
Traitement des erreurs....170
Tri....44
Tri de tableaux de chaînes....44
Txt 16/Text 8....20
Types de boucles....148
Types de variables....25

V

Valeur absolue....71
Valeur arrondie....74
Variable Byte....25
Variables à virgule flottante....26
Variables booléennes....25

W

Window library....281